KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Multi-view Architecture Description for Distributed Systems

Steven Op de beeck

Dissertation presented in partial
fulfillment of the requirements for the
degree of Doctor in Engineering

March 2014

Multi-view Architecture Description for Distributed Systems

Steven OP DE BEECK

Examination Committee:
Prof. dr. C. Vandecasteele, chair
Prof. dr. ir. W. Joosen, supervisor
Prof. dr. ir. E. Steegmans
Prof. dr. ir. T. Holvoet
Prof. dr. ir. F. Piessens
Dr. ir. M. van Dooren
Dr. B. Lagaisse
Prof. dr. V. Jonckers
(Vrije Universiteit Brussel)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

March 2014

© 2014 KU Leuven — Faculty of Engineering Science
Uitgegeven in eigen beheer, Steven Op de beeck, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

ISBN 978-94-6018-814-5
D/2014/7515/38

To Mona, Daan, and Sofie

Dankwoord

Graag wil ik iedereen bedanken die een professionele of persoonlijke rol heeft gespeeld in de realisatie van dit doctoraat. Allereerst bedank ik Wouter Joosen. Als promotor heeft hij me de kans en de vrijheid gegeven om m'n eigen pad te bepalen in dit onderzoek. Bedankt voor het vertrouwen tijdens de zoektocht naar een manier om dit werk te positioneren. *Kill your darlings* is soms de enige manier om vooruit te gaan.

Ik bedank ook Eddy Truyen die me als beginnend onderzoeker geen seconde tijd heeft doen verliezen aan het zoeken naar interessante en relevante papers. Bedankt ook voor de inspiratie die, enkele jaren later, de aanzet was tot dit werk. We waren het niet altijd eens, maar ik heb je mening altijd geapprecieerd, ook tijdens AOSD 2009 in Virginia. ;)

Ik wens ook de voorzitter, Prof. Carlo Vandecasteele, en de leden van de jury, Prof. Wouter Joosen, Prof. Eric Steegmans, Prof. Tom Holvoet, Prof. Frank Piessens, Prof. Viviane Jonckers, Dr. Marko van Dooren, Dr. Bert Lagaisse, te bedanken voor hun constructieve commentaar en vragen die mij toelieten om dit werk verder te verbeteren. Bedankt ook Marko en Bert voor hun complementaire invloed op dit werk. Er wordt wel eens lachend gezegd dat de aanhangers van (programmeer)talen en middleware lijnrecht tegenover elkaar staan. Dit werk toont aan dat het mogelijk is om ze te verzoenen.

Ook bedank ik Dimitri Van Landuyt voor onze onafgebroken samenwerking. We flirtten samen met verschillende paper deadlines en we werkten aan het e-Media systeem dat aan de basis ligt van dit werk. Ik denk dat hier complementair opnieuw het sleutelwoord tot een succesvolle samenwerking is. Daarnaast hebben we ook liters koffie de trappen af gedragen en uren gediscussieerd over maatschappelijke en andere topics. *Gratias tibi ago.*

Verder wens ik ook de leden van mijn onderzoeksgroep te bedanken voor hun kritische feedback en sociale afwisseling: Bart, Wouter, Stefan, Maarten, Jasper, Fatih, Ansar, Kris, Maarten, Frans, Johan, Adriaan. Ook bedank ik mijn vele bureaugenoten doorheen de jaren: Dominique, Thomas, Dimiter, Dries, Naeem, Ruben, Yves, Dries; Ansar, Julien, Andrew; Kurt, Danny, Robrecht, Koen, …, en daarnaast ook de grote groep collega's en ex-collega's waaronder Kim, Koen, Thomas, Davy, Sam, Aram.

Onderzoek is niet altijd even makkelijk te combineren met een lief, en later met een gezin. Creativiteit en de goesting om te schrijven kennen geen kantooruren, noch weekends. Bedankt Sofie voor je begrip, geduld, en om er altijd te zijn voor mij. Bedankt voor die schatten van kinderen, Daan en Mona. En sorry dat ik ook vanavond wat later zal thuis zijn.

Ik bedank ook mijn vrienden voor de nodige *R&R* na het werk, virtueel en *IRL*: Jan-Frederik, Koen, Michel, Wouter, en Michael; Dennis; en voor de leut tijdens en na de studies: Bram, Nelis, Dirk, Peter, en Kris.

En *last but not least* wil ik mijn ouders bedanken om mij de kans te geven de studies aan te vangen die aan dit werk vooraf zijn gegaan. Bedankt voor jullie steun en om in mij te geloven. Ook mijn zus mag ik niet vergeten.

—Steven Op de beeck
vrijdag, 14 februari, 2014

Abstract

Designing the software architecture is an essential part of the development of distributed systems. Software architecture is commonly defined as the set of structures, consisting of elements, their relations, and properties of both, that are needed to reason about the system. Architects specify these structures using multiple models, grouped into various architectural views based on the specific facet of the system they address. In the context of distributed systems the typical views are module, component-and-connector and allocation.

Yet, designing an architecture is not enough. It needs to be documented to be useful to developers. Documenting software architectures is a broad topic. In this dissertation we focus on architecture description languages (ADLs). Architects employ ADLs to describe architectural models in an accurate manner.

The problem statement of this dissertation is threefold. First, while the related work offers a wide field of ADLs that target analysis, verification, construction, and communication, most of these ADLs operate within the context of a single view. To the best of our knowledge, there is no ADL that targets the three views associated with distributed systems. Second, designing the software architecture of a distributed system is a highly iterative process. Architects address requirements and postponed decisions as part of a stepwise design process. Using state-of-the-art ADLs, this process results in monolithic models that have been contributed to by multiple architects in terms of multiple views at different times. Third, a lack of tool support decreases the usefulness of an ADL. Every ADL needs editing tools to assist in the creation of descriptions, and analysis or code generation tools to increase their usefulness once they have been defined.

As a first contribution, we present MViewADL, an ADL designed for distributed systems, that integrates the views of module, component-and-connector and allocation. The ADL supports the description of components and connectors,

the specification of their runtime instances, and the allocation of instances onto distributed hosts. The second contribution is called ReVew. It is a concept and technique for the modularization of the contributions of architects to the architecture description of a distributed system. The concept introduces the generic idea of stepwise refinement of architecture description. The technique implements the concept in a generic way, to allow its use in various ADLs. The third contribution is tool support for the production of MViewADL and ReVew descriptions, and subsequent code generation to multiple middleware platforms.

The validation of our work is twofold. The first part is an experience-based validation where we apply MViewADL and our tool support in a industry-grade case study on e-Media. The second part is a modularity and variability study where we show that ReVew is better able to contain the ripple effects of architectural changes, and demands less architect effort in realising eight variability scenarios, when compared to existing techniques for description modularization and reuse.

Beknopte samenvatting

Het ontwerpen van de software architectuur is een essentieel onderdeel van de ontwikkeling van gedistribueerde systemen. Software architectuur wordt in het algemeen gedefinieerd als het geheel van structuren, bestaande uit elementen, hun relaties en eigenschappen van beide, die nodig zijn om te redeneren over het systeem. Architecten specificeren deze structuren met behulp van meerdere modellen, gegroepeerd in verschillende architecturale views op basis van het specifieke facet van het systeem dat wordt beschouwd. In de context van gedistribueerde systemen zijn de typische views het module, component-en-connector en allocatie view.

Maar het ontwerpen van een architectuur alleen is niet genoeg. Een architectuur moet ook worden gedocumenteerd om van enig nut te zijn voor de ontwikkelaars van de software. Documenteren van software architecturen is een breed onderwerp. In dit proefschrift richten we ons specifiek op de architecturale beschrijvingstalen (ADL). Architecten gebruiken ADLs om hun modellen op een zo accuraat mogelijke manier te beschrijven.

De probleemstelling van dit proefschrift is drieledig. Ten eerste, ondanks de zeer uiteenlopende aard van het gerelateerd werk, situeren de meeste ADLs zich binnen het kader van één view. Voor zover wij hebben kunnen vaststellen, bestaat er geen ADL die ondersteuning biedt voor de drie architecturale views geassocieerd aan gedistribueerde systemen. Ten tweede, het ontwerpen van de software architectuur van een gedistribueerd systeem is een zeer iteratief proces. Een belangrijk onderdeel van dit stapsgewijs proces is het behandelen van nieuwe vereisten en uitgestelde beslissingen. Aan de hand van state-of-the-art ADLs resulteert dit proces in monolithische modellen waartoe wordt bijgedragen door meerdere architecten in termen van meerdere views op verschillende tijdstippen. Ten derde, een gebrek aan werktuigondersteuning voor ADLs tast de toepasbaarheid van de ADL aan

in het dagelijkse gebruik door architecten. Elke ADL heeft toepassingen nodig die de architect assisteren bij het aanmaken van beschrijvingen en toepassingen om het nut van deze beschrijvingen te verhogen eenmaal ze zijn gedefinieerd (bijvoorbeeld via analyse of code generatie).

In een eerste bijdrage, presenteren we MViewADL, een ADL specifiek ontworpen voor gedistribueerde systemen, dat de drie typische architecturale views (module, component-en-connector, en allocatie) integreert. De ADL ondersteunt de beschrijving van de componenten en connectoren, de specificatie van de instanties tijdens uitvoeringstijd en de allocatie van deze instanties aan gedistribueerde machines. De tweede bijdrage is ReVew. Het is een concept en techniek voor de modularisering van de bijdragen van architecten aan de architecturale beschrijving van een gedistribueerd systeem. Het concept introduceert het generieke idee van stapsgewijze verfijning van architecturale beschrijvingen. De techniek implementeert het concept op een generieke manier. Dit maakt het gebruik ervan in verschillende ADLs mogelijk. De derde bijdrage voorziet werktuigondersteuning voor de productie van MViewADL en ReVew beschrijvingen en de daaropvolgende code generatie naar meerdere middleware platformen.

De validatie van ons werk is tweeledig. Het eerste deel is een op ervaring gebaseerde validatie waar we MViewADL en onze werktuigondersteuning toepassen in een industriewaardige casestudie over e-Media. Het tweede deel is een modulariteit- en variabiliteitsstudie, waar we ons werk vergelijken met bestaande technieken voor modularisering en hergebruik van beschrijving. Hierin tonen we enerzijds aan dat ReVew beter in staat is om de effecten van veranderingen in de architectuur in te perken. Anderzijds tonen we aan dat het realiseren van deze acht variabiliteitscenario's minder werk betekent voor architecten.

Outline

1 Introduction **1**

2 A Multi-view Architecture Description Language **23**

3 Improved Modularity in Architecture Description **65**

4 Platform and Tool Support **117**

5 Conclusion **141**

A Syntax of MViewADL **149**

B Variability and Modularity Evaluation Raw Results **157**

Bibliography **163**

List of Publications **177**

Contents

1 Introduction **1**

1.1 Context 2

1.1.1 Software Architecture 2

1.1.2 Distributed Systems 10

1.1.3 Middleware 11

1.2 Related Work 12

1.3 Goals 15

1.4 Approach 16

1.5 Contributions 18

1.6 The e-Media Case Study 19

1.7 Overview 22

2 A Multi-view Architecture Description Language **23**

2.1 Goals 24

2.2 An Overview of MViewADL 29

2.2.1 Introduction 29

2.2.2 Interfaces 31

2.2.3 Components 35

2.2.4 Connectors 39
2.2.5 OO-Composition 41
2.2.6 AO-Composition 45
2.2.7 Configurations 52
2.3 Validation and Discussion 59
2.4 Conclusion and Future Work 62

3 Improved Modularity in Architecture Description 65
3.1 Introduction 66
3.1.1 Context 66
3.1.2 Problem Statement and Illustration 68
3.1.3 The Multi-view Refinement Solution 74
3.2 The Concept of Multi-view Refinement 75
3.2.1 Defining Refinement 75
3.2.2 Multi-view Refinement 76
3.2.3 Requirements for a Multi-view Refinement Technique . . 79
3.3 Related Work 81
3.3.1 Refinement and Inheritance 81
3.3.2 Analysis of Support in ADLs 82
3.4 ReVew — A Multi-view Refinement Technique 85
3.4.1 Refinement of Declarations 86
3.4.2 Redefinition of Members 89
3.5 Applying ReVew to MViewADL 91
3.5.1 The Interface 91
3.5.2 The Component 92
3.5.3 The Connector 93

3.5.4 The Application 98
3.6 Evaluation 101
3.6.1 Architectural Variations in the e-Media Case Study . . . 102
3.6.2 Evaluating The Impact of Variability on the Description Effort of Architects 111
3.6.3 Evaluating Modularity in the Change Scenarios 112
3.6.4 Conclusion of the Evaluation 114
3.7 Conclusion 115

4 Platform and Tool Support **117**
4.1 Goals 118
4.2 Maintaining Traceability in Implementation 119
4.2.1 Components and Interfaces 120
4.2.2 Connectors and Compositions 122
4.2.3 Platform-specific Strategies 128
4.3 An Overview of MViewADL Tool Support 131
4.3.1 The Basic Toolset 132
4.3.2 The Chameleon Workbench 132
4.3.3 The MViewADL Extension 133
4.3.4 Middleware Code Generators 136
4.4 Related Work 137
4.5 Validation 138
4.6 Conclusion 139

5 Conclusion **141**
5.1 Contributions 142
5.2 Future Work 145

A Syntax of MViewADL **149**

B Variability and Modularity Evaluation Raw Results **157**

B.1 Evaluation data for the change scenarios 158

B.2 Evaluation data for the ReVew technique 159

B.3 Evaluation data for the Import technique 160

B.4 Evaluation data for the Flatten technique 161

Bibliography **163**

List of Publications **177**

List of Listings

2.1 The `NewsBrowse` interface allows customers to browse and read news 33

2.2 The `@ServiceUsage` property attaches an additional semantic to a method 34

2.3 The `NewspaperService` component declaration with its require and provide parts 37

2.4 The `NSClient` component declaration only has a require part . . . 37

2.5 For customers to manage their consumer profile 38

2.6 For employees to manage the newspaper service 38

2.7 Various ways of charging customers 38

2.8 The `AccountingService` component has a provide and a require part 39

2.9 The `Content-` and `UserManagementSystem` components each provide a single interface 39

2.10 The implicit connector for the `NewsBrowse` interface 42

2.11 The connector functions as an adaptor between mismatching interfaces 43

2.12 The `NewspaperAccountingCn` *connector* consists of one *AO-composition* declaration, called `ServiceAccounting` 47

2.13 The `NewspaperApplication` aggregates various architectural elements 55

2.14 A fragment of the NewspaperApplication description that defines a physical topology 58

3.1 The NewspaperService component in MViewADL, combining two architect contributions into a single description 71

3.2 The ServiceAccounting AO-Composition in FractalADL, combining contributions from three expert architects into a single description 72

3.3 Abstract grammar for declaration refinement 86

3.4 The abstract component NSClient defines a generic client of the newspaper service 87

3.5 The MobileService component refines NSClient and inherits its require interfaces 87

3.6 Refining the NewsBrowse interface 92

3.7 A simplified NewspaperService component declaration 92

3.8 A refinement of the NewspaperService component that substitutes the provide interface 93

3.9 A refinement of the NewspaperService component with additional provide and require interfaces 93

3.10 The abstract ServiceUsageCn connector and ServiceUsage AO-composition 95

3.11 Refinement of a Connector and an AO-composition with an *advice* member 96

3.12 The NewsAccounting connector refining the ServiceAccounting connector with a component condition 97

3.13 The BasicNewspaperApp application declaration 99

3.14 The AccountedNewspaperApp refining the BasicNewspaperApp application with a host, instances and a connector 99

3.15 The NewsAccountingCn connector refining its parent with a host condition; in the context of the NewspaperAppDeployment. 100

3.16 The LoadBalancedNewspaperApp application description, refining the BasicNewspaperApp application with hosts and instances. . . . 105

3.17 The LB_AccountedNewspaperApp application description, refining AccountedNewspaperApp and LoadBalancedNewspaperApp with hosts, instances and connector refinements for load balancing. 107

3.18 The LB_NewspaperApp and NP_NewspaperApp declarations, combining three and two other apps through refinement, respectively.numbers 108

3.19 Application deployment descriptions for the BasicNewspaperApp in the CNN and NYTimes environments 109

3.20 The NYT_NP_NewspaperApp description, defining additional hosts and refining the Authentication connector. 109

3.21 The StagedNewspaperApp description, defining an additional host, a NewspaperService instance and the stem staging connector. . . 110

3.22 The NYT_NP_StagedNewspaperApp application, applying staging to the non-personalized NYTimes Newspaper application. 110

4.1 The essence of a component description lies in the interfaces it provides and requires 120

4.2 The NewspaperService JBoss component implements its contract by providing the NewsBrowse and requiring ContentBrowsing . . . 121

4.3 The Spring implementation of NewspaperService differs slightly in terms of annotations 121

4.4 A component that is refined with additional provide and require dependencies 122

4.5 Extending the component's contract with a provision of ConsumerProfile and NSManagement, and a dependency on UserProfiling 122

4.6 A JBoss connector implementation 126

4.7 Runtime Host Condition Evaluation in JBoss 127

A.1 Syntax of various types and modifiers, the identifier, and the qualified reference 150

A.2 Single and multiple refinement 151

A.3 The top-level elements are the interface, component, connector, and application 151

A.4 The interface declaration consists of methods 151

A.5 The method declaration 152

A.6 The component declaration consists of dependencies 152

A.7 The connector and optional AO-composition declarations 153

A.8 The pointcut declaration consists of its kind, signature and optional actor (caller/callee) definitions 154

A.9 The advice declaration consists of an advice method, the advice type and optional instance 155

A.10 The application declaration consists of host, instance and inline component and connector declarations 156

List of Figures

1.1 The Concepts of View, Viewpoint, and Model in context, as defined by ISO Standard 42010 [ISO10] 4

1.2 Software architecture has a direct impact on many stages in the development life-cycle 8

1.3 The Twin Peaks development model weaves together requirements and architecture (as illustrated in [Nus01]) 9

1.4 A deployment sketch of a digital publishing system 20

1.5 A model of the component-and-connector view on the e-Media architecture 21

2.1 The key that the element illustrations (interface, component, connector, application, etc.) in this chapter conform to 30

2.2 The components that participate in the running example, with their dependencies 31

2.3 Graphical representation that is equivalent to the MViewADL description of the `NewspaperService` component 36

2.4 A connector composing two components based on interface identity 41

2.5 Composition ambiguity when a required interface is provided multiple times 44

2.6 The graphical representation of the AO connector lacks the details of its MViewADL representation 51

2.7 Allocation of component instances onto abstract hosts 56

2.8 Allocation in a physical environment 58

3.1 Multi-view refinement of the `NewspaperService` component (left) and the `ServiceAccounting` connector (right) 77

3.2 Multi-view refinement of an MViewADL connector 94

3.3 ReVew refinement supports the structured description of the many variations of the e-Media system 104

3.4 The *Flatten* technique results in eight strongly overlapping variants in which the variations are hard-coded 106

3.5 ReVew demands less architect effort to realise all eight architecture variations *(left)* and, more specifically, demands a lot less effort of the deployment architect (*right*) 111

3.6 Containing change ripple effects with modularity: change affects fewer (or equal) descriptors when using ReVew *(left)* and fewer lines of descriptor code (*right*), across all three scenarios 113

4.1 Communication between remote aspects through a proxy component 130

4.2 The bold IDE building blocks we built ourselves 132

4.3 The third-party ANTLR tool allows us to design and generate a parser for MViewADL 134

4.4 Support for the typical code editing features like highlighting, code walking and error reporting 135

4.5 Generation of a JBoss Aspect from an MViewADL connector . . 136

CHAPTER 1

Introduction

"Any problem in computer science can be solved with another level of indirection. Except for the problem of too many layers of indirection."

— David Wheeler [Lam07]

Multi-view Architecture Description for Distributed Systems is concerned with the structural description of software architectures. It is architecture description that takes into account the various views of software architects that are essential in the development of distributed systems. In particular, we regard the well-known architectural views of module, component-and-connector, and allocation. Architecture description comes in many forms, with varying purposes. The goal of this work is to document the architecture with a focus on distribution, and in such a way that it facilitates iterative development. Furthermore, it targets descriptions that can be understood by developers, yet, that are formal enough to support code generation to middleware systems.

In this chapter we first present the wider context of the dissertation, which centres on software architecture in relation to middleware and distributed systems. We

continue with a study of the related work in architecture description languages. Based on the understanding of the context and the related work, we introduce the goals, approach and contributions of the thesis. We present the case study, that we have used to demonstrate the important research results of this work. We conclude with an overview of the dissertation.

1.1 Context

This section discusses the context of the dissertation. The broad focus of this work is architecture description for distributed systems with support for code generation to middleware systems. As a consequence, the context is situated in the topics of Software Architecture, Distributed Systems and Middleware.

1.1.1 Software Architecture

Software architecture is not that box-and-line drawing that is sometimes projected during meetings. Nor is it a system's earliest design decisions or its high-level structure. In fact, any of these things is merely a representation of the architecture that is relevant to a particular stakeholder. However, too often they lack the detail and the precision to be considered architectural at all.

We start with a definition of software architecture, followed by a discussion of what software architecture entails in terms of its key characteristics: *abstraction*, *architectural views*, *architecture description*, and its place in the *development process*.

A Definition

To get an understanding of what Software Architecture is, we take a look at some of the standard works in software architecture research that provide a definition of the concept:

Perry and Wolf. *"Software Architecture = {Elements, Form, Rationale}."* There are three classes of *elements:* processing, data and connecting elements. The *form* consists of element properties and relationships. The *rationale* denotes architect motivation [PW92].

Bass, Clements, and Kazman. *"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."* [BCK13].

Taylor, Medvidović, and Dashofy *"The set of principal design decisions governing a system."* Such decisions encompass structural concerns, such as components, connectors, and configurations; the system's deployment, non-functional properties, and evolution patterns [TMD09].

While these definitions all share the basic concepts of *elements*, *relationships* and *properties*, they do betray subtle differences.

What they all agree on is that a software architecture is something that can be observed and that it is not a phase of development. Moreover, every software system has an architecture and that architecture exists whether or not it has been documented. This means that if the knowledge of an architecture has been lost, it can technically be rediscovered by studying the system.

All works consider rationale (the reasoning behind a decision) to be part of the documentation of a software architecture. However, they disagree on whether it is essential in characterizing the architecture of a system. Bass, et al. compare rationale to the owner's manual for a car: it may help to understand the car, but it is not *part of* the car. The difference mainly results from a more pragmatic focus on software architecture by Bass, et al., while Perry, et al. and Taylor, et al. look at architecture from a foundational research angle. For the same reason, the formal angle in the latter two works is more prominent than it is in Bass, et al.

Based on these definitions one might be inclined to think that software architecture is only concerned with the structural nature of a system. This is not at all the case. While the design of an architecture typically starts with a structural decomposition into elements with well-defined responsibilities, relationships between these elements are defined that describe how the elements interact. These relations form the basis for the behavioural aspect of the architecture. A behavioural description of the system defines how interactions between elements may affect one another (e.g. the issues of deadlock and timing), and how they may affect the elements (e.g. changes to a component's state, or concurrency between components) [BBC+02].

This misunderstanding lies at the basis of why the box-and-line drawing is considered non-architectural: while it may suggest a structure of elements and relationships, it lacks the details (properties, responsibility, behaviour) that make such a structure architectural.

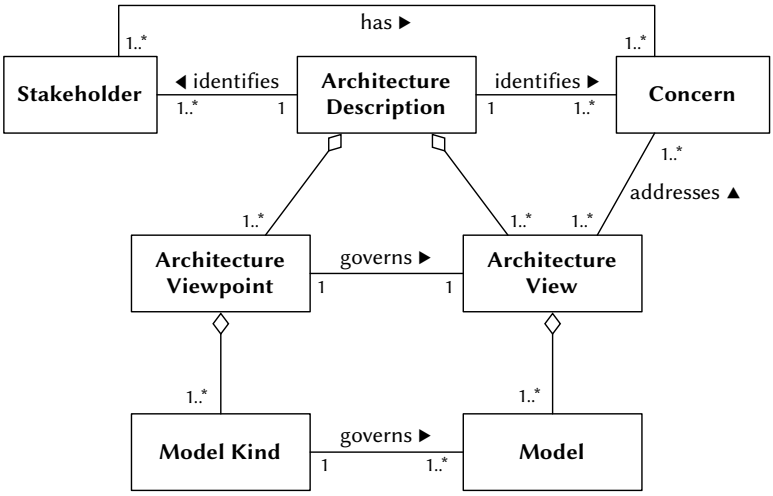


Figure 1.1: The Concepts of View, Viewpoint, and Model in context, as defined by ISO Standard 42010 [ISO10]

Abstraction

A common quality of architecture is that it embraces abstraction. Abstraction enables architects to reason about a system in terms of a particular set of elements, relations and their properties. Furthermore, abstraction allows the architect to focus on particular details of the elements while ignoring others. Abstraction is essential in the design of complex systems, as an architect cannot deal with all of the aspects of the system, all of the time.

Abstraction has many applications in architecture. One example is the interfaces of elements that separate the publicly visible properties from the private properties. It allows these elements to be developed in relative isolation, to be composed later on with other elements. In addition, the composition of elements is yet another example of abstraction. While elements focus on their behaviour, the relations between elements deal with composing this behaviour in terms of their public interfaces.

But, perhaps the most obvious form of abstraction in architecture is the *view*.

Architectural Views

The *view* is a form of abstraction that is typically associated with architecture. An architectural view captures the concerns of stakeholders on specific facets of the system.

In the literature and discussions on software architecture, the word 'view' has many different meanings. Sometimes it denotes a diagram consisting of a plethora of architectural elements —the box-and-line diagram, with one or more of these diagrams describing the architecture. Other times the term is used as a categorization of the types of diagrams that describe the architecture. Furthermore, the terms 'view', 'viewpoint', and 'perspective' are often used interchangeably.

Despite the apparent confusion around this concept, at least one of the earliest well-known interpretations of a view can be found in the 1995 paper titled *The "4+1" View Model of Software Architecture* [Kru95] by Philippe Kruchten. Kruchten proposes a view that addresses a specific set of architectural concerns in terms of a concrete set of architectural elements. In addition, Kruchten describes five different views: logical, process, physical, development, and scenarios, each one addressing one specific set of concerns, that allow to organize the description of a software architecture, when used concurrently. To capture the nature of a view, Kruchten employs the definition of software architecture by Perry and Wolf (Software Architecture = {Elements, Form, Rationale}):

> *We apply Perry & Wolf's equation independently on each view, i.e., for each view we define the set of elements to use (components, containers, and connectors), we capture the forms and patterns that work, and we capture the rationale and constraints, connecting the architecture to some of the requirements.*

An evolution of the view concept can be found in the work of Bass, et al. [BCK13] and is part of ISO standard 42010 on Architecture description [ISO10]. It is illustrated by means of the model in Figure 1.1 and can be summarized as follows: The description of, and reasoning about, the architecture of a system happens in terms of a limited set of views that each consist of one or more architecture models. Each view addresses a cohesive set of developer stakeholder concerns. Each model describes the architecture from the perspective of a particular view. The models in a view are created using the knowledge contained in predefined architectural

viewpoints. A viewpoint prescribes the stakeholders, typical concerns, model kinds, notations, elements, patterns, styles, etc. —*view : viewpoint :: map : legend*[1]

Based on this definition of the view, Bass, et al. [BCK13] propose a principal set of architectural views for reasoning about and describing the structure of a system —a refinement of earlier proposals for architectural views— that consists of the module, component-and-connector, and allocation view:

Module view models embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured.

Component-and-connector view models embody decisions as to how the system is to be structured as a set of elements that have runtime behaviour (components) and interactions (connectors).

Allocation view models embody decisions as to how the system will relate to non-software structures in its environment (CPUs, networks, teams).

Depending on the kind, size and complexity of a software architecture, an architect may not need all of these views, or he may define additional views to capture and reason about development concerns specific to his domain [RW11].

Architectural views constitute a frame for reasoning about software architecture that is built on best practices and past experience of the architectural community and the software architect.

Architecture Description

Remember that a software system has an architecture whether or not that architecture has been documented. Quality architecture documentation, also known as Architecture Description (AD), should be able to serve various purposes. Bass, et al. put forward three such purposes for AD:

1. As a means for education.
2. As a primary vehicle for communication among stakeholders.
3. As a basis for system analysis and construction.

[1] A view is to a viewpoint what a map is to a legend.

One could argue that all three purposes come down to communication with a particular group of stakeholders. Where the first purpose is targeted at communication with junior architects and analysts. The second purpose is aimed at architects, the customer, and other involved parties. The third purpose is intended for analysts and developers. While it may seem that AD is useful to a wide range of stakeholder, one thing that sets these stakeholder groups apart is the form of the AD. The form of AD ranges from informal (used for the first purpose and sometimes the second) to formal specifications (used for the second, but mainly the third purpose). Examples of informal specifications are diagrams or a deck of slides, while formal specifications use precise notations that are often referred to as Architecture Description Languages (ADLs).

An ADL is a formal notation that defines the concepts and semantics for the specification of architectural models. There exists a wide range of ADLs that have been proposed in academia as well as in industry [MLM+13]. Some of the early ADLs, like Wright [All97] and Rapide [Luc96] focus on analysing or simulating the dynamic behaviour of concurrent systems, while MetaH [BEJV96] is concerned with process scheduling. There's SADL [MR97b] for formal refinement of architectures across levels of detail, and more recently, π-ADL [Oqu04a] for formal modelling, analysis and refinement of dynamic software architectures, and AADL [FLVC05] which focusses on embedded systems for avionics and automotive applications.

This generic definition and the wide range of examples betrays the lack of consensus on what exactly constitutes an ADL. Medvidović, et al. describe ADLs as a spectrum [MR97a, MT00]. On the one hand, they argue, the goal of architecture description is to aid understanding and communication about a system, through simple, understandable, and possibly graphical syntax, well understood, but not necessarily formally defined semantics. While on the other hand, the tendency has been to provide formal syntax and semantics of ADLs, analysis tools, model checkers, parsers, compilers, etc. The authors content that both are important and should be reflected in an ADL.

As the frame for reasoning and description of software architectures, one would expect architectural views to be well supported in the state-of-the-art ADLs. Unfortunately, most ADLs focus only on a single view [WH05, Woo05, BWH+08], or more to the point, most ADLs operate within the context of one particular view.

Despite software architecture for distributed systems being the subject of study for more than twenty years, most practices, tools and notations have not yet made the leap from research to practice [BM07, Völ07], and those that are being used in industry, often originated there [MLM+13].

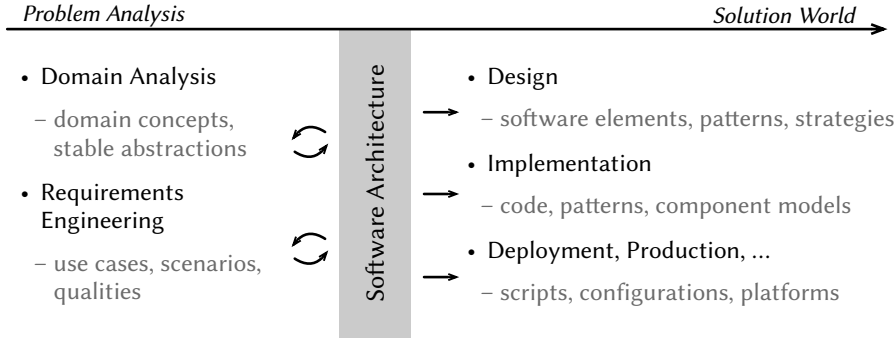


Figure 1.2: Software architecture has a direct impact on many stages in the development life-cycle

Development Process

It is too simplistic to define software architecture as a properly boxed up phase in the development life-cycle that turns requirements into a high-level design, further to be refined in the detailed design phase. Rather, software architecture acts as a bridge between the problem analysis phases and the phases where a solution is designed in detail, implemented, deployed, etc.

Figure 1.2 captures schematically how software architecture can be seen having a much more direct impact on the other stages in the development cycle than is the case in common understanding, which is still influenced by the waterfall model [Roy87, NL03, LN04]. In the figure we make a distinction between problem analysis on the one hand and the solution world on the other [Jac01].

Domain Analysis [Nei80, PD90, Ara94, JBR99, Fow02, Kru03, Eva04, Som10] is the process of analysing related software systems in a domain to find their common and variable parts. Domain analysis and its main result, a model of the domain, are considered an important factor in the success of software reusability. In addition, the use of stable domain abstractions in a domain model has been linked to effective reuse in literature [MHF+97, Som10, Kel06], i.e. in a component-based context, components implementing a stable domain abstraction are more likely to be reusable. Stable domain abstractions are defined as "fundamental concepts in the application domain that change slowly". In a realistic development setting, the problem domain is often the only common ground between the different developers and therefore, stable domain abstractions form an important

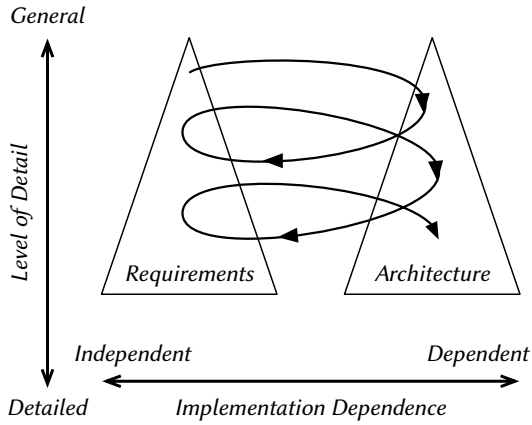


Figure 1.3: The Twin Peaks development model weaves together requirements and architecture (as illustrated in [Nus01])

means of communication between the different stakeholders. In domain-driven design [Eva04], the set of stable domain abstractions is also called a *ubiquitous vocabulary*.

Requirements engineering (RE) is defined as the process of discovering the purpose for which a software system is intended, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation [NE00]. RE consists of the activities of *elicitation*, *analysis*, *modelling* and *evolution*. Elicitation is the driving activity in the process. It deals with identifying the stakeholders [SFG99], and obtaining an understanding of their goals [DvLF93], or the tasks they currently perform using, for instance, scenarios [Mai98] or use cases [Jac87]. Analysis is responsible for discovering conflicts, handling trade-off and performing consistency checking, verification, validation, etc. Some of these analysis techniques require a specific modelling of the requirements, while communication with stakeholders often requires a less formal form. Finally, evolution is all about managing (long-term) change in requirements, and understanding the impact on the developed system.

The Twin Peaks model for software development [Nus01] considers an intertwining of requirements engineering and architecture creation, leading to a more incremental development process. This incremental process, as sketched in Figure 1.3, is almost essential when we consider that requirements are not static in nature. They change over time, even during the creation of architectural solutions.

This is a consequence of the trade-off that occurs in development. The impact of a trade-off between two solutions may have an effect on the requirements that are concerned with these solution.

At the other side of the architectural bridge in Figure 1.2 are the phases of (detailed) design, implementation, deployment, etc. Each of these phases is directly impacted by the software architecture. Architecture defines the boundaries that contain the detailed design. Architecture constrains algorithms and other solutions in the implementation. Architecture ensures qualities like availability and performance by guiding allocation. The software architecture has an impact in each of these phases in the development life-cycle.

1.1.2 Distributed Systems

A distributed system, in essence, is a system of hardware or software elements that perform coordinated tasks by means of communication through a network. Distributed systems are certainly not the only systems that benefit from the proper documenting of their software architecture. However, the particular works that we have referred to in the software architecture literature often consider architecture in the context of distributed systems and include material that deals with deployment and networked nodes. The reason for this is that distributed systems are more challenging and versatile in terms of the architectural qualities that need to be supported. The development of distributed systems opens the door to challenges such as availability, concurrency, failure handling, heterogeneity, transparency, scalability, etc.

This meant that, before long, experts were looking at ways to improve the development of large distributed systems that became increasingly capable and complex. As this happened at roughly the same time that research into software architecture took off [Sha89], it is not surprising that distribution concepts are so ubiquitous in software architecture literature. Distribution became one of the many complexities of software systems that architecture was tasked to help manage.

Today, not only has computing become ubiquitous as a consequence of ever growing networks, the networks themselves have become pervasive as well, as buildings, vehicles and individuals are being equipped with increasingly powerful and complex devices that communicate (often) wirelessly.

Impact on Software Architecture. Perry, et al. [PW92] already refer to multi-processor and multi-host architectures in their software architecture foundations paper. Kruchten's early work [Kru95] on architectural views has a *Physical view* that deals with concurrent processes, networked hosts, and the mapping between both.

The more recent literature on the topic includes distribution as an essential part of the description of a software architecture. In Bass, et al. [BCK13] the *Allocation view* considers deployment on networked nodes. In addition, their work considers architecture description in the presence of familiar issues like concurrency, availability, communication, physical allocation, etc.

Taylor, et al. address the distribution aspects of software architecture in their work on Foundations, Theory and Practice [TMD09]. They have chapters on deployment and mobility, application of architecture and various distribution styles.

The presence of a physical or allocation view might suggest that this view is solely responsible for capturing the distributed nature of a system. This is not the case. While allocation is certainly essential in this regard, distribution has an impact throughout a system's architecture and its description. More specifically, it can be found in concepts such as loose coupling between components, substitutability and heterogeneity of components and connectors, adaptation, communication, concurrency, availability, scalability, etc. Distribution is an essential part of Software Architecture.

1.1.3 Middleware

Developing distributed software systems is a complex task which demands a systematic approach at every stage of development. While the overall availability and performance of a distributed system is laid out during software architecture, a lot still depends on the quality of the implementation of that system. It is the task of middleware to raise the level of abstraction to allow developers to focus on the issues of availability, performance, consistency, integrity, etc., instead of basic functionality such as distribution. Middleware offers programming abstractions and services that handle basic needs like the communication between multiple processes across a network, persistence, transactions and security primitives.

Industry is actively involved in research and development of various middleware platforms, ranging from embedded middleware for sensor networks [HMCP04,

GH09], over message-oriented middleware for SOA applications [Cur05], like IBM's WebSphere [IBM13], and Redhat's [Red13b] and Oracle's [Ora13b] SOA Platforms, to general-purpose application middleware [SSA02], like DyMac [Lag09], GoPivotal's Spring [GoP13], RedHat's JBoss Application server [Red13a], and Oracle's Application Server [Ora13a].

In this work the focus is on application middleware platforms like Pivotal's Spring Framework and RedHat's JBoss AS, where architecting distributed applications involves *complex compositions* of components and third-party subsystems. This complexity is inherent to the growing need to take into account the runtime and distribution characteristics of compositions, as well as their crosscutting nature. This is a trend that is gaining support from AO-Middleware platforms (AOM) such as ReflexD, DyMAC, and AWED [TN05, LJ06, NSV+06], which offer direct support for such complex compositions.

1.2 Related Work

Software architecture has long been concerned with the distributed nature of systems. As a consequence, a good number of architecture description languages exists that supports particular aspects of distributed systems. Specifically, we consider those languages that focus on the structure (Darwin, LEDA, SADL, Olan, AADL, π-ADL, DAOP-ADL, AO-ADL, Fractal-ADL, AspectLEDA, AspectualACME, and Prisma), and behaviour (Wright, Rapide, Darwin, LEDA, and π-ADL) of distributed systems. In addition, we also consider languages that support multiple views (AADL), and that offer techniques for iterative development (SADL, π-ADL, and AADL).

ADLs like Wright and Rapide focus on formally describing the behaviour of a system for the purpose of architectural analysis. Wright is tailored to validating consistency and completeness (e.g. deadlock detection) in concurrent systems [ADG98] while Rapide models ordered events sets in dynamic systems [Luc96]. Both ADLs model interfaces and components, but only Wright has explicit connectors and configurations, and supports the creation of component and connector instances. Connections in Rapide are highly dynamic links between required and provided interfaces. Neither approach considers the explicit description of allocation to networked nodes. Both languages consider structural and behavioural descriptions within the context of the Component-and-connector view.

Darwin and LEDA allow the description of dynamically reconfigurable distributed systems from hierarchical combinations of components. π-ADL is a general-purpose language for the architecture description of structure and behaviour of dynamic systems. Darwin, π-ADL, LEDA (and Wright, Rapide) all employ some level of indirection in binding components. Darwin [MDEK95] consists of (hierarchical) components and their provided and required services that are linked by binding them together. π-ADL [Oqu04a] on the other hand consists of components, connectors and configurations of both. Components and connectors are bound together on the basis of their ports. π-ADL also considers behaviour of components and connectors. Instantiation is left implicit. LEDA [CPT99] considers (hierarchical) components and the roles that describe and model their behaviour. LEDA does not distinguish between components and connectors, instead components are bound on the basis of roles. All three use π-calculus for modelling the semantics of structures and/or behaviour. Darwin has a concept of worker nodes or logical processes, but does not make distribution nodes explicit [MDK94]. π-ADL nor LEDA support the explicit description of allocation to networked nodes. The descriptions of Darwin, LEDA and π-ADL are situated within the Component-and-connector view.

SADL is a formal refinement language [MR97b]. It allows the specification of high-level properties that must be satisfied by subsequent refinements of the architecture. The language models components, connectors and configurations. Components are bound together on the basis of the ports they define (indirection). Components and connectors are types, but instantiation and allocation is not considered. Without the explicit presence of instances, SADL focusses on Module view descriptions of components and connectors.

Olan is a configuration language for the abstract description and deployment of the implementation of distributed applications [BBB$^+$98]. The language supports abstractions such as interfaces, components, and composite components. A component is mapped to its implementation artefact based on its interfaces. Components are grouped together into a composite component, based on common properties like a shared allocation host. Olan is tied closely to implementation and its primary stakeholders seem to be programmers, not architects.

Then there's AADL that focusses on the interactions between processes in embedded systems [FLVC05]. It models systems of processes and threads that make use of resources such as processors, memory and devices. While we do not consider AADL to be distributed, it is of interest to use as it considers very detailed allocations of elements onto logical (or physical) nodes. AADL is situated across the Component-and-connector and Allocation views.

Finally, there is a body of fairly recent work in the field of aspect-oriented ADLs. All these languages, DAOP-ADL [PFT05], AO-ADL [PFT11], Fractal-ADL [PSDC08], AspectLEDA [NPTM09], AspectualACME [GCB+06], and Prisma [PRM+03] use a combination of components, connectors, configurations and sometimes aspects to model the system. However, because of their specific focus on AO, they tend to be less concerned with the description of distribution in the software systems that they model.

Discussion

We discuss the related work in terms of general distribution concepts, such as loose coupling, component substitutability and heterogeneity, runtime instances, and allocation to networked hosts.

Distribution. As most ADLs consider dynamic systems, the binding of components is often performed using some mechanism for indirection or loose coupling. The mechanisms that we have come across in the related work are ports, services, interfaces, events, and connectors. The mechanisms are not exclusive, multiple of these can be used in combination.

Less than half of the ADLs we studied provide explicit support for components and connectors at the instance level. Of course, if the goal is to model a specific aspect of the architecture, e.g. events in the system [Luc96], this is not necessarily a problem. Nevertheless, component (and connector) instances are an important aspect in the development and description of distributed systems.

Allocation. Some of these ADLs support the specification of so called configurations of components (and connectors), while others only structure descriptions through recursive hierarchies of components. Since we are dealing with distributed applications, it is not unimportant to consider the degree of distribution beyond the granularity of the component. Not every component is necessarily its own island, i.e. a network node is able to support multiple components.

If we consider the architectural views that these ADLs conform to, then most of them target Component-and-Connector exclusively. Some, arguably, also consider the Module view, as they define static component types that expose interfaces. As far as we are aware, no ADL considers topologies of hosts and the abstract and/or physical allocation of instances to these hosts. Olan, an

implementation-level configuration language, considers groups of components that can define an attribute that names the deployment node. AADL describes allocation to system resources, instead of networked hosts, in addition to a process model.

Summary. Distribution is a very broad topic. Every ADL we studied characterizes itself as a description language for distributed systems, yet no two languages are completely alike in focus and design. There are a fair number of ADLs that are focused on the analysis or verification of a specific part of the architecture. That these languages do not consider a broader view on the structure of the system, is often by design. Despite this broad field of interest, the concern of runtime instances and particularly allocation is still underserved. Similarly, in terms of support for architectural views, most ADLs operate within the Component-and-connector view. As far as we know, there is no ADL that considers network distribution in terms of the Allocation view.

One of the conclusions from a very recent and sizeable survey on *"What Industry Needs from Architectural Languages"* by Malavolta, Lago, et al. [MLM+13], is that among the most important features for ADLs, tool support, support for iterative architecting [MDT07], and support for multiple architectural views stand out on a set of twenty features that practitioners were asked to order on a scale from *definitely useful* to *definitely not useful*.

1.3 Goals

This work has been driven by a set of objectives that are summarized into the following three goals:

1. *Support developers (architects, designers, programmers, deployers, etc.) in the creation of distributed software systems.* By combining our experience in distributed systems with our growing knowledge of software architecture, we know (a.) what is important to the developers of such systems [OdbVLTJ08, VOTV10], and (b.) we have learned that architecture is a suited way to capture the information that they require [VOTJ12, OvDLJ13]:

    a. Indirectly coupled, heterogeneous elements, that can be connected by means of properly designed, expressive and descriptive compositions;

a runtime understanding of elements and compositions; and a description of how the system relates to its environment.

b. Architecture description that can capture the inputs, or views, of various architectural experts (module, component-and-connector, allocation), including the concepts and abstractions in (a.).

2. *Increase the separation of concerns in the models produced using architecture description languages.* Software architectures are developed by means of an iterative process that considers additional requirements and postponed architectural decisions in each iteration. Currently, this results too often in monolithic models that are being contributed to by multiple architects, at various times. Our approach is to structure these contributions by applying various separation of concerns techniques. We will also attempt to keep our solution as generic as possible, so it does not unnecessarily limit the range of ADLs to which it can be applied.

3. *Build tools to further support development.* This includes tools for the creation and verification of models created by architects, as well as tools for the generation of implementation artefacts for various distributed middleware platforms, to assist developers.

1.4 Approach

This work started on the basis of two assets, which we had built before: an extensive literature study of *separation of concerns* techniques in Requirements, Architecture, Design and Implementation [OG05, OTB+06, OGTJ06], and an industry-strength case study and demonstrator on e-Media.

The subject of our case study is a content distribution system for e-Media: a digital newspaper that offers news and various additional services (e.g.: billing and personalized content). The case study had been developed in close cooperation with the Flemish publishing industry over the course of a couple of years [VLGM+06, MJVL+07, VOK+]. It has been used —and continues to be used— as the case study in a number of papers [BWH10, SBJ10, BSJ09] and courses, and the concepts and techniques that were developed for the case study have been the subject of several publications [VOTJ09, VOTJ12, OdbvDLJ12, OvDLJ13] and demonstrations at various venues, such as Middleware'08 [OdbVLTJ08], AOSD'09, Summer School on AOSD'10.

The e-Media case study has also served this research as a source of inspiration, illustration, validation, and experimentation. It was essential in the creation of an environment that was complex enough to reveal the problems that we set out to solve. These complexities include non-trivial behaviour and interaction of the components in the system that were captured, in part, by techniques like aspect orientation. The scale of an e-Media platform promoted thinking about caching and load-balancing which had an effect on system deployment. And creating an implementation of the system introduced us to the real-life complexities of persistence, transactions, security, composition and packaging in middleware frameworks.

One thing the e-Media case and our literature study have in common is a focus on the full development life-cycle, ranging from requirements engineering to deployment. While most of the contributions of this work are situated in software architecture, this regard for the full life-cycle is still present in our efforts to couple architecture descriptions to implementation and deployment, and the tool support that we provide for architects and developers.

Like in software architecture, continuous refinement played an essential role in the development of our work. Because the three goals that we put forward in this research are very much connected, progress in one goal (*"dimension"*) has an impact in the other two. The first goal is related to improving the abstractions that an architect has at her disposal (what) while the second goal is concerned with improving how the architect uses these abstractions to describe an architecture (how), and the third goal is concerned with providing tools that support the solutions to goals one and two. So, for instance, adding an abstraction, like a host definition, to the part on architecture description requires that we figure out how the *host* behaves under iterative development of the architecture description, and how we implement it in our tool.

Looking back, while tool support was expected to be a validation of our research, in reality, it was also a driver that helped the research progress as we prototyped ideas, instead of heading off in the abstract. The non-academic nature of the e-Media case study, in combination with our work on tooling and live demonstrations, resulted in contributions that are a mix of experience and academic rigour.

1.5 Contributions

The contribution of this dissertation is threefold. The first contribution is a multi-view ADL for distributed systems, called MViewADL. The second contribution is a stepwise refinement technique for ADLs, called ReVew, that offers support for the iterative development of software architectures. The third contribution consists of a number of smaller realizations, including code generation and tool support for MViewADL and ReVew.

1. **A multi-view ADL for distributed systems.** We introduce MViewADL, an ADL for the description of distributed systems that integrates the heterogeneous expertise of architects. The ADL supports the modelling of loosely coupled elements (i.e. components) that are related through interface dependency. It has expressive connectors that support adaptation between mismatching interfaces, and that allow reasoning about the runtime and distributed nature of the application (AO-composition). Last but not least, MViewADL supports a mapping of elements onto a layout of networked nodes. As such, the ADL supports concepts from the architectural views of Module, Component-and-connector, and Allocation. (Chapter 2.)

2. **Iterative development in ADLs.** We define ReVew, a concept and technique for the modularisation of the contributions of architects to the architecture description of a distributed system. The concept introduces the generic idea of stepwise refinement of architecture description. The technique implements the concept in a generic way, to allow its use in various ADLs. We evaluate ReVew by applying the technique to MViewADL and by validating it in an extended case-study. The validation shows that ReVew increases modularity and variability in architecture description. (Chapter 3.)

3. **ADL tool support and code generation.** To support the two main contributions of this dissertation, a number of additional results have been realised. First, we have created a set of tools to support MViewADL and ReVew. These tools served a number of purposes: they assisted us in the definition of our case study, and they allowed us to verify and fine-tune both the language as well as the refinement technique. Second, we have implemented code generation to two middleware platforms (JbossAS and Spring). This allowed us to verify the generic nature of the ADL, and it increases the practical usefulness of the language. (Chapter 4.)

1.6 The e-Media Case Study

In this section we define the e-Media case study that is used throughout the dissertation. The subject of our case study is a content distribution system for e-Media. This case study was developed throughout a number of research projects in the e-Media space, in close collaboration with partners in the Flemish publishing industry. The architecture and implementation were created in the context of an AOSD Industry Demonstrator [VOTV10]. The case study serves as a way to validate our research, and as a source of inspiration for real-world problems in distributed systems.

Overview. The way in which news is produced and distributed is changing significantly. The business landscape of publishing is drastically evolving towards a more competitive environment with a lot of new players emerging from unexpected sides such as Google News. This has put a lot of pressure on the traditional publishing companies to evolve to more competitive news offerings.

There has been a trend towards digital publishing platforms that support multiple service offerings which are bundled depending on the targeted group of customers. A digital publishing platform offers news in different media formats and sizes, to be delivered through different telecommunication channels, and to be displayed on different end user terminals and devices. In this way, the daily (online) newspaper is complemented with an aggregation of differentiated services. These services may also offer personalized content and advertisements, tailored to the interests of individual customers, their location, situation and behaviour. Although it may not be clear yet whether this kind of flexible service offering represents the future of news publishing, it is certain that publishing companies will gain a competitive advantage if their ICT infrastructure can rapidly evolve towards supporting these, or other, kinds of offerings.

A digital newspaper offers news in different media formats and sizes, to be delivered through different communication channels, and to be displayed on different end user terminals and devices. It supports various additional services, like flexible accounting, tracking user-interest and content personalization. Personalized content and advertisements are tailored to the interests of individual customers, their location, situation and behaviour. A consumer can browse recent headlines and read article summaries for free. However, to have access to the full content and additional services, a consumer is required to sign up. The system charges a signed-up consumer for paid services by means of micro-payments.

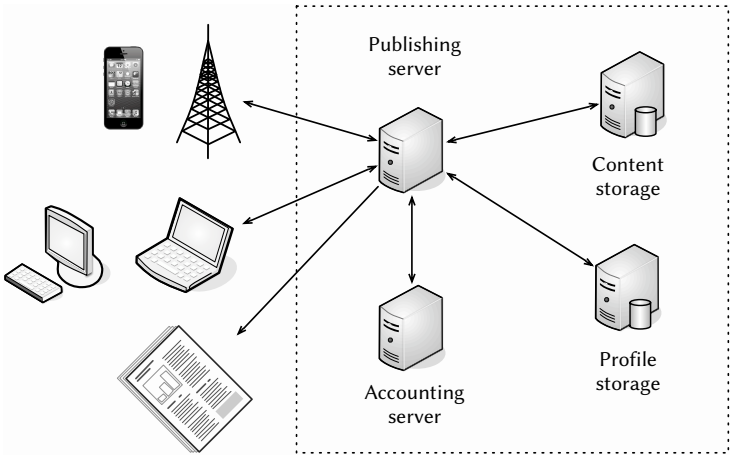


Figure 1.4: A deployment sketch of a digital publishing system

The deployment sketch in Figure 1.4 depicts a digital publishing platform. At the top-left side of the figure, the different output channels are presented: the publishing platform is able to deliver news in real-time to mobile users connecting on the move using a mobile device (through a telecom infrastructure). Also, the news services are offered to home users that connect via the publisher website. The publishing platform is even backwards-compatible in the sense that classical delivery mechanisms like printing and physical delivery of the newspaper are still supported. At the top-middle of the figure, the publishing server is presented. This node consist of the actual services offered by the publisher. At the bottom side of the figure, the accounting system is represented, connecting the publishing servers to financial institutions (e.g. bank servers) to trigger payment transactions. At the right-hand side of the figure, two separate back-end servers are represented, one for each of the main types of data. The news content consists of articles, advertisements, and their meta-data. The profile information consists of all customer information, together with customer's interests that are both static (defined by the user himself) and dynamic (inferred by the publishing system).

Architecture. Figure 1.5 shows a model of the component-and-connector view of a relevant subset of the publishing software architecture. The subset consists of component instances, their compositions and information on how they are allocated onto logical hosts.

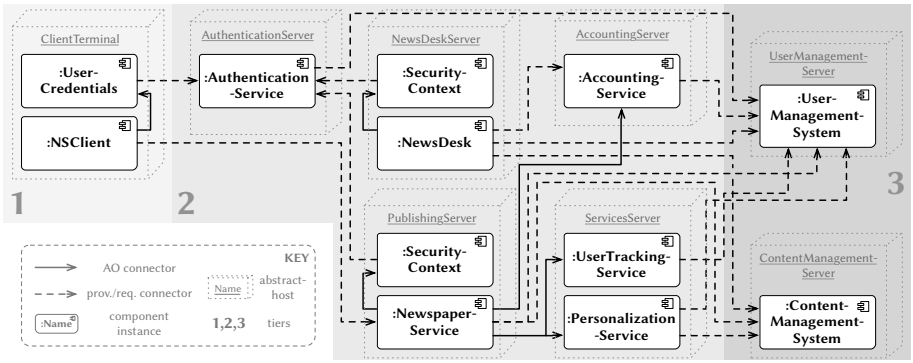


Figure 1.5: A model of the component-and-connector view on the e-Media architecture

The architecture is deployed along three tiers: (1) Client, (2) Business and (3) Storage.

The Business tier consists of a number of subsystems: the Newspaper, NewsDesk and Auxiliary services. The `NewspaperService` component is externally accessible by the `NSClient` component. It supplies the services to browse and read articles.

Authentication and Authorization are supplied by the `AuthenticationService`, `UserCredentials` and `SecurityContext` components. The additional services `UserTrackingService` and `PersonalizationService` can be activated on a per-user basis. User tracking keeps track of the reading behaviour of the consumer, while personalization uses this to personalize the view of the consumer on the news.

The `AccountingService` component is responsible for doing the accounting of *Service Usage* behaviour of the consumer. The methods that belong to this *Service Usage* category are fetchArticle, listNewestArticles, listArticlesForTag, and listArticlesForCategory —a part of the main interface of the `NewspaperService`.

Interactions in the model are represented by the full and dashed arrows between component instances. They are the connectors in the component-and-connector views. The difference between the full and dashed arrow is that the full one represents an ao-composition, while the dashed arrow represents a regular provide-require dependency between components.

While this overview is only concerned with a subset of the e-Media architecture, even that is too large for a proper running example. To this end, we will focus on

the NewspaperService, the AccountingService and the interaction between both components for most illustrations in this chapter. When we consider application assembly and host allocation, we will take into account the additional components and the logical host definition.

1.7 Overview

This dissertation is structured as follows.

Chapter 2 introduces MViewADL. It redefines the goals for our solution and revisits the related work in ADLs. It describes our solution by defining its concepts and by discussing specific challenges that needed solving. It ends with a discussion of the validation and revisits our goals in a conclusion.

Chapter 3 introduces ReVew, our concept and technique for improving the modularity in architecture description using ADLs. It analyses the related work based on a set of requirements and describes and illustrates the problem statement further. Our proposed solution is described in a generic and an applied manner. An evaluation compares our technique from a variability and modularity standpoint. And finally, a conclusion revisits our goals and analyses ReVew based on the earlier requirements.

Chapter 4 describes our tool support for MViewADL and ReVew. Support consists of code generation to the JBoss and Spring middleware platforms, and an Eclipse plug-in for the creation and verification of MViewADL descriptions.

Finally, Chapter 5 summarizes the contributions of this dissertation, and outlines directions for future research.

CHAPTER 2

A Multi-view Architecture Description Language

"In practice the use of such notations is rare."

— Bass, Clements, Kazman
on the topic of ADLs [BCK13]

Software architects use Architecture Description Languages (ADLs) to capture the structure and behaviour of software systems for reasons of documentation, verification or analysis. No single ADL is suited to describe every kind of software system in all its aspects. Therefore, various ADLs exist that target specific architectural aspects (correct architectural refinement, dynamic reconfiguration of systems, deadlock detection, process simulation, etc.) or particular types of software systems (dynamic systems, service-oriented architectures, embedded real-time systems, etc.).

In this chapter we present MViewADL, an ADL for the structural specification of distributed middleware systems. MViewADL focusses specifically on what it means for applications to be distributed (loose coupling, substitutability, heterogeneity, adaptation, allocation, etc.) and it acknowledges that software architects reason about distributed systems in terms of multiple views (the Module, Component-and-connector, and Allocation view). MViewADL is designed around the typical architectural concepts, the *interface*, *component*, *connector*, and *configuration*.

We start this chapter with an introduction of our goals for MViewADL (Section 2.1). For the related work of this chapter we refer back to the related work in the introduction of this dissertation, Section 1.2. The body of this chapter is an overview of MViewADL (Section 2.2). Finally, we discuss the experience-based validation of MViewADL (Section 2.3) and end with a conclusion (Section 2.4).

2.1 Goals

When designing and implementing a distributed system, developers depend on the software architecture and the architects to provide input for their activities. The concerns and questions that different developers have that should be answered by the software architecture include: Which components are assigned to what developer? What are their responsibilities and which behaviour do they export? How will these components interact to build a concrete (sub)system? How should the system be deployed? Developers may also have many more questions, such as how to handle security, availability, performance, recovery, etc.

Goal 1. Concepts of Distributed Systems

The first goal of MViewADL is to support developers (architects, designers, programmers, deployers, etc.) in the creation of distributed systems by providing software architects with a structured way to document the answers to some of these questions. More specifically, MViewADL focusses on describing the structure of a system in terms of the following architectural aspects:

1. Indirectly coupled and heterogeneous components, composed by
2. Expressive and descriptive connectors with a runtime understanding,

3. The assembly of a system from these components and connectors, and
4. The deployment of that system into a distributed environment.

In essence, MViewADL targets an integrated architecture description that connects activities from *defining components* to *allocating them on distributed hosts*. As such, its descriptions involve the knowledge of multiple architectural views such as the Module, Component-and-connector, and Allocation view.

Descriptions in MViewADL are structured in terms of elements such as interfaces, components, connectors, and configurations [All97, Wol97]. Taking into account the architectural aspects for distributed systems, as defined above, results in the following requirements, categorized by the applicable element. These requirements have been inspired by the state-of-the-art and -practice in middleware systems [Red13a, GoP13, TN05, NSV+06, LJ06] and the broad field of ADLs in the related work.

1. **Components**
   - □ Unique interfaces as semantic groups of related methods.
   - □ Heterogeneous components with provided interfaces that are loosely coupled based on interface dependency.
2. **Connectors**
   - □ Connectors that capture composition between components that is based on satisfying their dependencies, with support for adaptation between components with mismatching interfaces, and the substitutability of component alternatives.
   - □ Connectors that support expressive composition with a non-invasive join point model that allows reasoning about the runtime and distribution context of components (AO-composition).
3. **Configurations**
   - □ Configurations that describe a runtime system in terms of concurrently executing component instances, and connector instances that specify how these instances interact.
   - □ Configurations that describe the distributed layout of physical nodes and how component and connector instances are allocated on these nodes.

We do not claim any of the previously mentioned aspects of MViewADL as contribution in themselves. Indeed, as the state-of-the-art in ADLs is varied in terms of focus and design —as concluded in our study of the related work (Section 1.2), it is not hard to find ADLs that support some of these aspects in one form or another. Yet, we are not aware of any ADL that supports all aspects in an integrated manner.

Woods and Hilliard, reporting on the *Architecture Description Languages in Practice* workshop reached a similar conclusion [WH05]:

> *"All the ADLs that the group were aware of only support a single view, whereas one of the research ideas that practitioners have actually embraced is the use of multiple views."*

In an earlier paper Woods notes [Woo05]:

> *"Most of the ADLs appear to focus on describing the functional and/or concurrency structure of the system. I haven't discovered one that places similar emphasis on information or deployment structure, both of which are key concerns for information systems architects."*

The reasons for this trend are unclear. It has been suggested that the apparent rift between the languages and tools that researchers provide and the solutions that practitioners are looking for [Woo05, WH05, BM07, Völ07, MLM+13], is responsible. Or maybe it is related to another conclusion of the workshop: that most ADLs focus on being as generic as possible, eschewing any form of domain specialization.

MViewADL does not shy away from focussing on a particular domain, namely, distributed component-based middleware systems. Having this focus means that we know exactly how components are defined, how applications are assembled and how systems are deployed. It is clear that supporting multiple views in a single ADL introduces strong constraints. This leads to a trade-off between domain-specific languages where the overlap at the view-boundaries is clearly understood, and, more generic languages that target strictly independent descriptions. An example of the latter is UML, which supports multiple views by means of different model types with very strict boundaries. UML allows models with virtually no semantics that would limit their applicability. An anonymous reviewer of our work summarized it as follows:

> *"The different views are constrained to strongly overlap—they are not as independent as they might be in a pure ADL with no implementation support—but this is probably inevitable and in fact desirable given the goals of the [work]."*

While the first goal considers the individual elements of the multi-view description of distributed systems, the second goal of MViewADL deals with the definition of a frame for an integrated language that supports multi-view description. We define it as the need of the language to *integrate the contributions of architects with a heterogeneous expertise* (module developer, application assembler, deployer). The contributions of architects to the architecture description make use of the concepts and abstractions that match their individual expertise. For instance, a component developer will not consider instantiation or allocation of the components, while a deployer will not be redefining the interfaces of the components he allocates.

Goal 2. Integrating the Heterogeneous Expertise of Architects

An ADL should permit the architect to describe the structure of a system using abstractions that fit his intuition and vocabulary [All97]. The development of distributed systems adheres to a structure that consists of a number of development stages and coupled developer stakeholder roles [AF01, Szy02]:

1. a *module developer*, developing components and connectors
2. an *application assembler*, assembling applications from these modules
3. a *deployer*, deploying the system onto a physical infrastructure

At the architectural level, these stages are related to the architectural views of *Module*, *Component-and-connector*, and *Allocation* [BCK13], respectively. The Module view describes modules and the relations between these modules, such as *dependency*, *is part of*, etc. In this context, a module defines the basis of a component, while a relation can be captured using a connector. The Component-and-connector (C&C) view describes how components and connectors interact to form a runtime system. Where the Module view defines component and connector types, the C&C view defines runtime instances of these types. Instances are grouped into an application and allocated onto an logical host topology that

is organized into tiers. Finally, the Allocation view describes how the logical topology maps on concrete hosts in a physical environment.

A software architect that is developing a distributed system, in the context of a particular stage, assumes the same role as the expert stakeholder for that stage. The architect reasons about the design in terms of the view that matches his role. For instance, during application assembly, the architect will reason in terms of concepts of the Component-and-connector view, about which components and connectors to combine for the particular application, how many instances are required for each, which components may or may not reside together on the same host, etc. Another architect constructs interfaces, components, and how these components interact. Yet another architect reasons about the hosts in the physical environment.

At different times, architects with different roles are involved with the design of the architecture. Each of these architects is concerned with describing the architecture in terms of concepts of the architectural view that matches their expertise or their current role. Instead of creating multiple description languages, each one tailored to a specific view, we target one integrated description language that covers the relevant concepts of these three views.

The Multi-view Nature of Descriptions. When considering the architectural description of a particular element, for instance a connector, in terms of multiple views, this can have two meanings.

1. Architects use this element in descriptions of the system from the perspective of multiple views. An architect describes the connector in a model of the Module view, another one considers its instances and logical allocation in a model of the Component-and-connector view, and one specifies the physical hosts, where its instances are to be allocated, in a model of the Allocation view.

2. Architects consider the description of the element itself from the perspective of these various views. An architect describes a connector that captures an AO-composition in terms of the concepts in the Module view (interfaces, components), another architect extends the AO-composition with concepts of the Component-and-connector view (instances, allocation), etc.

In the first case, the element plays a role in various models from the perspective of different views, while in the second case, the description of one element integrates

concepts of multiple views. In addition, both meanings of multi-view description may be applicable at the same time.

The structure of the goals in this chapter mirrors that of the related work in Section 1.2: (1) *Concepts of Distributed Systems*, which discusses general distribution concepts in ADLs, and (2) *Integrating the Heterogeneous Expertise of Architects*, which deals with multi-view support in ADLs.

Throughout the development of the ADL, a key concern has always been to target the creation of models that can be understood by many, instead of just the architects that created them. This has had a big impact on the kinds of description that we have considered for MViewADL, avoiding notations such as XML.

2.2 An Overview of MViewADL

This section explains MViewADL and discusses the important issues we addressed. We start with an introduction that provides an overview of the section.

2.2.1 Introduction

As an architecture description language that describes the structure of distributed systems, MViewADL is built around the core architectural concepts of *Interface*, *Component*, *Connector*, and *Configuration*. The language has elements by the same name: *Interface*, *Component*, *Connector*, representing each of these concepts. The role of the configuration concept is an exception, it will be played by the *Application* element.

This chapter is structured around these concepts. We start the explanation of each concept with a short overview of the state-of-the-art and its representations throughout the development process. We describe the language element that represents this concept in MViewADL and how it relates to the other elements in the language. We illustrate this description using examples from the e-Media system.

Throughout these sections, we keep in mind the goals of MViewADL that have been put forward in Section 2.1. More specifically, we focus on the issues of distribution and various architect views in our explanation of the ADL.

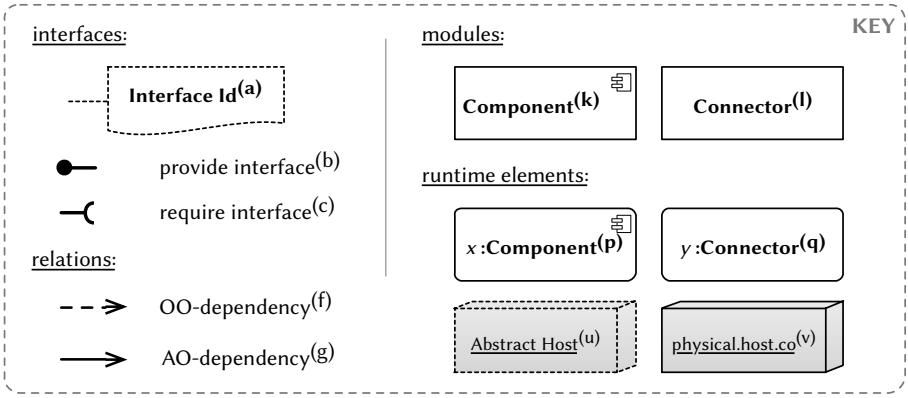


Figure 2.1: The key that the element illustrations (interface, component, connector, application, etc.) in this chapter conform to

The Interface, Component, Connector and Application elements are further addressed in Sections 2.2.2, 2.2.3, 2.2.4, and 2.2.7, respectively. Sections 2.2.5 and 2.2.6 address the OO and AO composition mechanisms supported by the connector.

Illustrations

The illustrations of the elements throughout this chapter all comply with the key in Figure 2.1: The *Interface Id* element (a) indicates the identity of the *interface* that it is attached to. An interface plays either a *provide* or a *require* role in its relation with a module element. The provide interface (b) is represented by a filled circle, while the require interface (c) is represented by a half circle.

Interfaces are the basis for dependencies between elements. A *usage* dependency between two elements is actually a *dependency* relation between the interfaces of both elements. Although it is allowed to make abstraction of the interfaces and represent dependencies between elements directly. There are two kinds of dependencies, an *object-oriented dependency* (f) and an *aspect-oriented dependency* (g).

There are two kinds of module elements: the *component* (k) and the *connector* (l). Both have a runtime representation: the *component instance* x (p) and the *connector instance* y (q), respectively. Instances are allocated onto hosts. Hosts can be *abstract* (u) or *physical* (v).

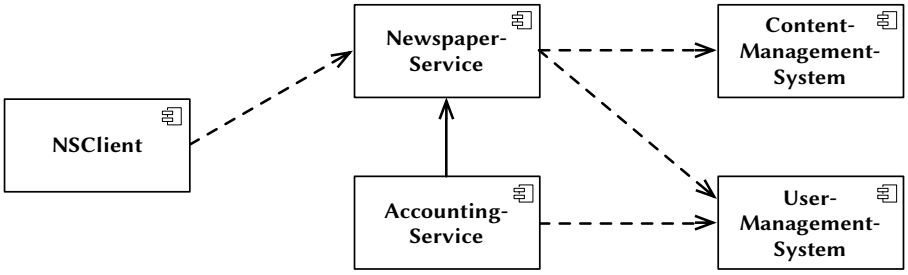


Figure 2.2: The components that participate in the running example, with their dependencies

The Running Example

The MViewADL language is explained by means of a running example from the e-Media case study. We focus the example on a specific subset of the publishing system: (a) the customer browsing and retrieving articles, and managing her consumer profile; (b) the publishing system that registers the customer's consumption of articles for billing purposes.

Figure 2.2 shows the components that participate in the running example, with the dependencies between these components. Scenario (a) of the subset involves the `NSClient` component calling the `NewspaperService` component. In turn, the `NewspaperService` needs the `Content-` and `UserManagementSystem` components to realize its behaviour. Scenario (b) involves the `AccountingService` component that intercepts particular calls to the `NewspaperService`. The `AccountingService` requires the `UserManagementSystem` to store the information it has registered.

In what follows, we will further detail these components and their interactions, by introducing the interfaces, connectors and compositions that specify these components and interactions. Furthermore, we will introduce the context in which these components run, by specifying their configuration in terms of instances and the allocation onto hosts.

2.2.2 Interfaces

Interfaces are essential in the definition and the description of components and connectors in ADLs. An interface can be considered an access point to a service

that a component provides. An interface consists of a coherent set of methods that describe the contract for interaction with the component. These methods belong together according to some arbitrary semantic grouping.

Interfaces are designed with a specific degree of granularity in mind, balancing number and size. However, this balancing act is always a trade-off, as it is impossible to capture every semantic group within a single decomposition. A component can provide multiple services, therefore it can have multiple interfaces that provide access.

Consider for example, in the context of the Newspaper Service in the e-Media case study, a set of methods for browsing and fetching articles and a set for managing ones subscription to the newspaper service. A defensible solution is to design an interface, `News`, to interact (browse and read) with the news, and to design another interface, `Profile`, to handle subscriptions. The difference in semantics between these two interfaces is obvious. Next, consider that some of the methods in the `News` interface require a subscription (read), and some do not (browse). This is clearly another semantic difference between the methods in this set. This time, however, the methods are already part of a single interface. The trade-off in this case is between splitting up the `News` interface —which is often undesirable, ignoring this particular semantic difference, or marking it in some formal or informal way. In most modern programming languages this scenario is supported using a combination of annotations and reflection or aspect-oriented techniques.

Interfaces in the Life-cycle. An interface is a fundamental yet basic concept. Its form does not change much throughout the software development process. An interface remains a description with a unique name that lists the services, operations, or methods that define its purpose. The syntax and the exposed detail of the methods (parameter types, error handling, etc.) may change depending on the context where it is used.

In software architecture, the interface plays a key role in the description of components and the dependencies between components in a system. However, some ADLs make abstraction of interfaces by representing component interactions by means of roles or ports.

```
interface NewsBrowse {
   ContentItem  fetchArticle (ContentItemId contentId);
   ContentItem  getShortVersion(ContentItemId contentId);
           Tag  getTag (String tag);
      Category  getCategory (String cat);
      List<..>  getSubCategories (Category category);
      List<..>  listNewestArticles (int amount);
      List<..>  listHeadLines (int amount);
      List<..>  listArticlesForTag (Tag tag);
      List<..>  listArticlesForCategory (Category category);
      //...
}
```

Listing 2.1: The `NewsBrowse` interface allows customers to browse and read news

2.2.2.1 The Interface Element

The interface element in MViewADL conforms to the typical *object interface* in component-based systems [Szy02]. An interface has an identity, represented by a name that is often a reflection of the methods it exposes. The identity of an interface is unique inside the system in which it is defined. This means that there is no other interface that carries that name.

Listing 2.1 shows an example of an interface description in MViewADL. This interface is named `NewsBrowse`. It contains all the methods that allow a customer to browse and read the news in the e-Media system.

An interface declaration has a name and consists of a set of zero or more method definitions. These methods should be abstract, ie. they are not allowed to have a body. All methods are considered public. The method does not diverge from the typical method definition in programming languages: it has a name, arguments and a return type.

Interfaces determine where the line is drawn in terms of modularity and information hiding of the component. They represent the ports that control access to the component. An interface plays a role in its relationship with a component or a connector that is either *require* or *provide*.

The unique identity of an interface, in combination with the role it plays in relation to a component, forms the basis of component composition. Only those components are composed that have matching interfaces with inverse interfaces roles.

```
interface NewsBrowse {
    @ServiceUsage ContentItem  fetchArticle (ContentItemId contentId);
                  ContentItem  getShortVersion (ContentItemId contentId);
                          Tag  getTag (String tag);
                     Category  getCategory (String cat);
                     List<..>  getSubCategories (Category category);
    @ServiceUsage    List<..>  listNewestArticles (int amount);
                     List<..>  listHeadLines (int amount);
    @ServiceUsage    List<..>  listArticlesForTag (Tag tag);
    @ServiceUsage    List<..>  listArticlesForCategory (Category category);
    //...
}
```

Listing 2.2: The `@ServiceUsage` property attaches an additional semantic to a method

2.2.2.2 The Property Interface

In addition to the object interface, MViewADL supports something we call a *property interface*. The property interface is essentially an extension to the *object interface*. It enables annotating the methods of an object interface, or the interface itself, with one or more *semantic properties*.

Listing 2.2 shows an example of a property interface, called `@ServiceUsage`, and how it is specified as an extension to a component interface. The semantic property expresses an additional characteristic of that method in addition to the semantics contained in the name of the method and its parameters, and the name of the interface. This allows for an unlimited number of semantic subgroups within the typical interface.

Referring to the semantic property, means referring to each method of every object interface that carries that property. Reasoning about a property interface is also limited to the semantic group as a whole, instead of each individual method. In MViewADL, a property interface can be refered to in the pointcut of an AO-composition. Unlike with an object interface, it is not possible for a component or connector to express their dependencies in terms of a property interface.

The property interface is an architectural expression of our previous work on stable abstractions for pointcut interfaces [VOTJ12]. Stable abstractions reflect fundamental concepts in designers' minds and communications that change slowly. Our research has shown that these abstractions are well suited for the purpose of defining *pointcut interfaces* [GK01, GSS+06].

2.2.3 Components

"Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system," is how Clemens Szyperski summarizes the component in his work on Component Software [Szy02]. While this description may appear to focus on runtime qualities, software components play an important role in the design, the development, and the deployment of software systems. In our work, we focus on the aspects of design and deployment of components in the context of software architecture. However, we will not ignore *executability* and *composition*.

A component shares many of the characteristics that lead to effective modules, including:

- Maximizing cohesion and minimizing dependencies, as proposed by Constantine [SMC74]
- Information hiding, as identified by Parnas [Par72]

In addition, a component has the following characteristics that distinguish it from the module [Szy02]:

- A component is a unit of composition.
- A component has explicit context dependencies.
- A component has no observable state.

As a *unit of composition*, it is in the nature of a component to be combined or composed with other components. Because components are units, they cannot be partitioned and must be composed in their entirety.

A component is often not fully self-contained. Whenever it uses the services of other components, it must explicitly *declare its dependencies*. Each component does this by specifying all of the interfaces that it requires in addition to those that it provides.

A component has *no observable state*. As such, a component (the client) that is using the services of another component (the server) can only depend on the well-defined interfaces of that component. The client component has no means of accessing the internal state of the server component itself. This does not mean that

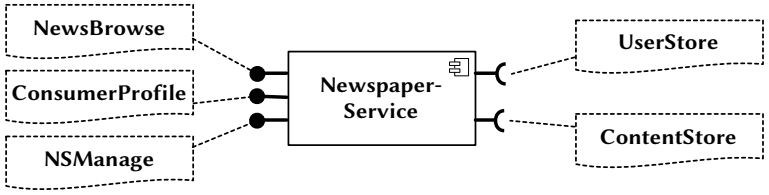


Figure 2.3: Graphical representation that is equivalent to the MViewADL description of the `NewspaperService` component

a component cannot have internal state in its implementation, only that it is not available or observable to external entities. The reason for having no observable state is the requirement that a component should be indistinguishable from copies of its own. This also does not mean that a certain component cannot provide access to the persistent state of the application (entries in a database, etc.). Rather, it should be transparent to the client which copy of the component is serving the request for data.

Components in the Life-cycle. Definitions of the term component in the state-of-the-art, like the following from a 1996 workshop on European Conference on Object-Oriented Programming, often include the concept of *independent deployment*:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

A component that has the property of independent deployment, is designed as a separate entity with respect to other components and its environment. While this is an important property —one that builds on the characteristics of the component— deployment is only one aspect of the component that is addressed during development. Hence why is not part of our main list of component characteristics.

A component is not solely a runtime (or design-time) element. Throughout the software development process, a component takes on many forms: an abstract modular unit that encapsulates behaviour behind an interface, a coding unit consisting of classes, interfaces and platform descriptors, a software element that

```
component NewspaperService {
    provide {
        NewsBrowse,
        ConsumerProfile,
        NSManage
    }
    require {
        UserStore,
        ContentStore
    }
}
```

Listing 2.3: The `NewspaperService` component declaration with its require and provide parts

```
component NSClient {
    require {
        NewsBrowse,
        ConsumerProfile
    }
}
```

Listing 2.4: The `NSClient` component declaration only has a require part

conforms to a component model, a platform-specific artefact that is executable by a specific middleware, etc. In the software architecture process alone it appears in many views, each using a particular configuration of components to describe an aspect of the system. However, in each of its forms, the component is delineated by the same set of characteristics.

2.2.3.1 The Component Element

In MViewADL, a component description consists of a *provide* and a *require* part. These describe the dependencies of the component in terms of the *interfaces* that it provides and requires, respectively. Both parts are optional. A component does not need a *require* part if it is completely self-contained. Although not mandatory, it is common for a component to have at least one interface in its *provide* part, as a way for other components to interact with it.

The example in Listing 2.3 shows the `NewspaperService` component with its dependencies. It provides three interfaces: `NewsBrowse`, `ConsumerProfile` and `NSManage`, and it requires two interfaces: `UserStore` and `ContentStore`. Figure 2.3 shows a graphical representation of the `NewspaperService` component with its five interfaces.

The `NSClient` component in Listing 2.4 only has a require part. It depends on two interfaces: `NewsBrowse` and `ConsumerProfile`. Both are provided by the `NewspaperService`.

```
interface ConsumerProfile {
    UserId createProfile(ContactInfo cInfo, PaymentInfo pInfo, String
        password);
    void setAdditionalInfo(Gender gender, Float yIncome, MaritalState
        mstate);
    MediaConsumerProfile consultProfile();
    Period getSubScriptionDetails();
    void setSubScriptionPeriod(Period period);
    //...
}
```

Listing 2.5: For customers to manage their consumer profile

```
interface NSManage {
     List<..>  getSubCategories(Category category);
     Category  getCategory(String categoryName);
          Tag  getTag(String tagName);
         void  addCategories(Set cats, Category parent);
         void  addTags(Set<String> tagLst);
         void  deleteCategory(Category parent, Category toDelete);
         void  deleteTag(Tag tag);
     List<..>  listArticles(Tag tag, int amount);
         void  addArticle(ContentItem cItem);
         void  deleteArticle(ContentId cId);
     //...
}
```

Listing 2.6: For employees to manage the newspaper service

```
interface AccountUsage {
    void chargeForSubscription(Period period);
    void chargeForService(Service service);
    void chargeForUsage(ContentItem item)
}
```

Listing 2.7: Various ways of charging customers

```
component AccountingService {
    provide { AccountUsage }
    require { UserStore }
}
```

Listing 2.8: The `AccountingService` component has a provide and a require part

```
component ContentManagementSystem {
    provide { ContentStore }
}
component UserManagementSystem {
    provide { UserStore }
}
```

Listing 2.9: The `Content-` and `UserManagementSystem` components each provide a single interface

The `ConsumerProfile` interface, in Listing 2.5, allows customers to manage their consumer profile. It has methods to create a profile from the necessary contact and payment information, and to manage the subscription period for access to the content. The `NSManage` interface, in Listing 2.6, is for newsdesk workers (employees) to manage the newspaper service. It provides methods for adding, browsing and deleting tags, categories and articles.

Our running example consists of three more component definitions: `AccountingService`, `Content-` and `UserManagementSystem`. The `AccountingService` component (Listing 2.8) provides an interface for registering a customers' usage of the newspaper service: `AccountUsage` (Listing 2.7). The final two components, `Content-` and `UserManagementSystem` (Listing 2.9) both provide only a single interface: `ContentStore` and `UserStore`, respectively. These interfaces are not further detailed here.

We will explain how these components are composed together in the next section, when we discuss the connector.

2.2.4 Connectors

If components are made for composition, then connectors are the glue that composes them. Yet, composition is a vague term, in the sense that it does not exactly convey how the whole is made up from its constituent parts. Composition of components comes down to the components working together to make up the whole system. However, there are multiple ways in which components can work together. Instead of just calling it composition, we need to look further, at the mechanism behind the composition: the connector.

Computation vs. Communication. Where the component is considered to be about *computation*, the connector is about *communication*. While computation and communication might appear to be two completely separate functions, this distinction between component and connector is only skin-deep. Both are part of a continuum, where the difference between their function is relegated to the role played by an underlying piece of implementation at a particular level of abstraction [Kel07].

For instance, consider a component, `NewspaperService`, that is developed on top of a lightweight component middleware. When this component requires the services of another component, `ContentManagementService`, it will call a particular method, the result will be computed, and returned to the caller —computation and communication. Now, say the middleware supports a distributed component model and the `ContentManagementService` is located remotely. What was previously a simple method call, is now a far more complex remote invocation that needs to travel through multiple layers of middleware services (marshalling, discovery, remote invocation, synchronization, …) before reaching the other component. Depending on the level of abstraction, we are dealing with a connector between two depending components (communication), a complementary set of marshal and unmarshal operations (computation), a component-lookup operation (computation), etc.

Computation and communication are two sides of the same coin. However, while the line separating these concepts may fade when observed from a distance, the component and the connector each play a specific role at a particular abstraction level. Consequently, while the component and the connector declarations in MViewADL may be similar in appearance, they play an entirely different role in the architecture description of a system.

Connectors in the Life-cycle. Similar to the component, the connector is not limited to a single representation during the software development process. However, unlike the component, the connector is not always a clearly identifiable software element. During architecture description, it can be a first-class element, a nameless line in a box-and-line diagram or it can be left implicit. During implementation, a connector can be a complex piece of code, a simple method call, an annotation, or it may not be implemented at all.

In software architecture, the form of the connector really depends on the level of abstraction at which the system is being considered. Once a trivial connection between components is evolved into a non-trivial composition, architects benefit

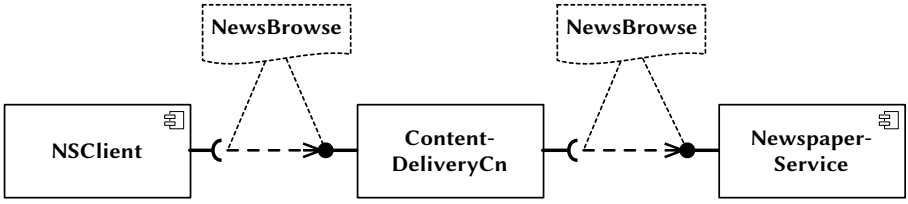


Figure 2.4: A connector composing two components based on interface identity

from the use of explicit connectors. It increases the independence of components by restructuring the way a component interacts with other components in the system [All97]. An explicit connector permits architects to capture the variable part of the composition separately. This benefits the reusability and adaptability of both the component and the connector.

The Connector Element

A connector in MViewADL describes the composition of components. In our ADL, components and connectors are related in the sense that they are both considered a module. This means that, like the component, a connector consists of a *provide* and a *require* part. The details of what it means for a connector to require and provide interfaces, depends on the composition mechanism that the connector employs.

The MViewADL connector offers two mechanisms for component composition: *object-oriented composition* (OO-composition) by means of coupling the required and provided interfaces of components, and *aspect-oriented composition* (AO-composition) which is based on the interception of component interactions by means of pointcut and advice. A particular connector can only support one composition mechanism at any time.

We discuss OO-composition in Section 2.2.5 and AO-composition in Section 2.2.6.

2.2.5 OO-Composition

A connector that employs the OO-composition mechanism composes two matching components by connecting them such that one component satisfies the dependency of the other. Consider the `ContentDeliveryCn` connector in Figure 2.4.

```
connector _implicit_NewsBrowseCn {
    require { NewsBrowse }
    provide { NewsBrowse }
}
```

Listing 2.10: The implicit connector for the `NewsBrowse` interface

In this example, the `NSClient` component *requires* the `NewsBrowse` interface, while the `NewspaperService` component *provides* this interface. The `ContentDeliveryCn` connector is straightforward. It *provides* the `NewsBrowse` interface to the `NSClient` component, hereby resolving the dependency of the client, while *requiring* the `NewsBrowse` interface itself from `NewspaperService` component. So, instead of offering the provided behaviour of the `NewsBrowse` interface itself, like a component would, it merely acts as a bridge to the component that does provide this behaviour.

Specifying connectors like this may seem pretty straightforward, however there are a number of issues that we still need to address:

1. Describing these connectors is a tedious job that does not scale well for larger systems.
2. Composing heterogeneous components that do not share identical interfaces.
3. Dealing with the ambiguity of an interface that is provided by multiple component.

We discuss each of these problems in the subsections that follow (2.2.5.1 – 2.2.5.3).

2.2.5.1 The Implicit Connector

Components define their dependencies in terms of the identities of the interfaces they *require*. An architect creates a connector for each required interface of every component to couple it to a provided interface of another component. However, in the common case when there is only one component that provides a certain interface and one or more components that require that interface, that connector is implicit and may be left undefined.

Listing 2.10 shows an example of such an implicit connector. It composes the `NewspaperService` and `NSClient` components in Listings 2.3 and 2.4, respectively.

```
connector ContentDeliveryAdaptorCn {
    require { NewsBrowse }
    provide { ContentBrowse }
}
```

Listing 2.11: The connector functions as an adaptor between mismatching interfaces

However, the composition that is specified here can be resolved automatically by matching the identities of the provided and the required interfaces of the involved components.

2.2.5.2 Composition of Components with Mismatching Interfaces

When an architect needs to compose independently developed components with mismatching interfaces, a connector can be defined that acts as an adaptor between both components. Such a connector requires a different interface than the one it provides.

For instance, suppose the interaction scenario between the client component and NewspaperService component. The client requires a `NewsBrowse` interface, while the NewspaperService provides a similar, but not fully compatible `ContentBrowse` interface. To have both components interoperate, the connector needs to prescribe the conversion behaviour that maps the one interface onto the other. The example in Listing 2.11 shows the `ContentDeliveryAdaptorCn` connector which adapts the composition between the mismatching interfaces `NewsBrowse` and `ContentStore`.

Adapting compositions cannot be deduced automatically, and must always be explicitly specified. As a general rule, a connector must always require the behaviour it provides, as long as it does not require complex computations that are better handled by a component.

2.2.5.3 Ambiguity in OO-Compositions

What happens when there is a component that requires an interface that is provided by at least two different components? Figure 2.5 illustrates this ambiguity visually. The `ContentDeliveryCn` connector is pictured as having a dependency on the `NewsBrowse` interface. However, as there are two components that provide

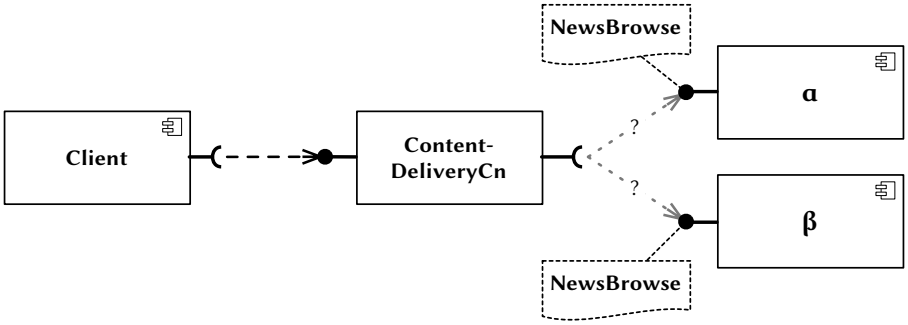


Figure 2.5: Composition ambiguity when a required interface is provided multiple times

this interface, α and β, it is unclear which of the two will be selected for composition. We know that the interface in question, although provided by multiple components, is one and the same, as there can be only one interface by its identifier.

Unfortunately, the architect that tries to uniquely describe this ambiguous composition will fail. Because the architect cannot select which component to compose to, he cannot solve the ambiguity.

The only option at this time would be to disallow two or more components to provide the same interface. However, this would compromise the substitutability of components, where multiple alternative components can provide the same interface.

Since there is no other solution at this time, we will postpone solving it until we are describing the system at the instance level in the Configurations section (Section 2.2.7). As it turns out that this situation is similar to the ambiguity of having multiple runtime instances of the same providing component.

2.2.5.4 Interface Dependency as the Basis of Composition

An interesting and tempting idea is to support connectors the can be related to other connectors. This would allow a train of connectors to be created that composes two components. One reason to support this is that a fairly complex connector could be decomposed into many simpler connectors, that would work

together. However, the reason that this is not supported in MViewADL is twofold. First, the ADL would deviate from the state-of-the-art middleware systems that we target with our architecture description. The benefits of connector chaining would have to be carefully weighed against the increased complexity of the models and of the translation into the subsequent development activities (implementing, deploying, etc.). Second, it would lead to connectors that are less concerned with *communication* and increasingly concerned with *computation* (conversion and processing of data). The current way to do this is to interleave components into the chain of connectors that handle the computation, and leave the connectors to handle the communication[1].

The way composition works in the ADL, and how the architect is able to specify it, is through the mapping and prior definition of required and provided interfaces. The architect has no direct control over which components are getting composed to which connectors. The only thing he can specify is the dependencies of these components and connectors, by listing the interfaces they provide and require. How these elements will eventually be related is up to the semantics of the language, and later on, the language runtime which implements these semantics. The language semantics state that the only way to connect two components is by means of a single connector, based on the dependencies that these elements carry.

2.2.6 AO-Composition

The AO-composition technique differs from OO-composition in that it is not about matching the required and provided interfaces of components. Rather, AO-composition supports the interception of the interactions between those components that are composed using either composition mechanism. An example of an interaction is one component calling a method of a provided interface of a component it is composed to. This interception technique used in AO-composition is based on the concepts of pointcut and advice.

In state-of-the-art AO-middleware [Lag09] AO-composition is marked by the following properties:

1. **Declarative composition specifications.** As a declarative composition specification, the AO-composition is a structured description instead of an

[1]The reason we mention this idea is that it allows us to point out that this is not possible in MViewADL; and to further explain why it is not.

imperative procedure: architects model the pointcut and select the advice method using MViewADL notations.

2. **Non-invasive and distribution-aware join point model.** A *non-invasive join point model* implies that the kind of join points that can be advised in an MViewADL pointcut is limited to the elements in the interfaces of components and the context of these components, with properties such as the interfaces it provides, and which component it is. As such, the internal details of components cannot be advised.

   In a *distribution-aware join point model*, the context of a join point includes properties such as the exact *component instance* that has been intercepted, what *application* the component is part of, and onto which *host* it is allocated. In case these elements all seem unfamiliar, they are part of the component-and-connector and allocation views, and will be explained in Section 2.2.7, where we describe the Application element.

3. **Composition of components.** This characteristic states that AO-composition is, just as OO-composition, concerned with the composition of components —as opposed to for instance the composition of components with aspects. To properly explain this characteristic, a deeper understanding of the structure of the pointcut and the advice is necessary. We will come back to this property near the end of the AO-composition subsection.

A connector in MViewADL may contain zero or more AO-composition elements. However, a single connector can express either an OO-composition or an AO-composition, but not both. So, once the connector contains an AO-composition element, it can no longer express OO-composition. An AO-composition description consists of two parts that can only appear once in the description: the *pointcut* element and the *advice* element. Being part of the same AO-composition definition is what binds a specific pointcut to a specific advice. We describe these two elements and the example, further in what follows.

Listing 2.12 shows an example of an AO-composition, called `ServiceAccounting`, that is contained within the `NewspaperAccountingCn` connector. The goal of this connector is to compose the accounting service (specified in the advice) to the newspaper service (specified in the pointcut).

```
connector NewspaperAccountingCn {
  require {
    AccountUsage
  }

  ao-composition ServiceAccounting {
    pointcut {
      kind: execution;
      signature: @ServiceUsage;
      callee {
        interface: NewsBrowse;
        component: NewspaperService;
        host: PublishingServerA, PublishingServerB;
      }
    }
    advice {
      type: after;
      method: chargeForUsage(..);
    }
  }
}
```

Listing 2.12: The `NewspaperAccountingCn` *connector* consists of one *AO-composition* declaration, called `ServiceAccounting`

2.2.6.1 The Description of a Pointcut

A pointcut is a structured expression that defines the join points and conditions that determine where and when the interception of an existing component interaction should take place. While pointcut expression in MViewADL is not fundamentally different from the related work, our language has a unique focus. MViewADL has specifications that are extensible by different architects with another expertise, and it supports reasoning about the distributed nature of a system.

In MViewADL, the pointcut is expressed in terms of the *kind* and the *signature* elements, and at most two *actor* elements. The actor elements enable reasoning about the join point context of the caller and callee components of an intercepted interaction. Therefore we will call these actor elements by their specific instances: the *caller* and the *callee*.

The kind, signature, caller, and callee members of the pointcut are related through conjunction. We say in this work that adding specifications for the kind, signature, and caller and callee elements further *constrains* the pointcut, making it more specific with each additional property specification.

The Kind Element. The first pointcut property refers to the kind of join point. It has two possible values, either `call` or `execution`. The pointcut kind refers to the two join point variations by the same name, the *call* and *execution* join point. Consider the interception of a component interaction: the `call` kind indicates interception at the side of the component that requires the behaviour, while `execution` indicates the side of the component that provides the behaviour. `Execution` is the default value.

An example of the kind element is shown in listing 2.12 (line 8), where the kind of the `ServiceUsage` AO-composition pointcut is set to `execution`.

The distinction between call and execution is important in distributed systems, because it determines the flow of join point context information, between the two components involved in the intercepted interaction, that is required to validate all caller and callee conditions. Since this flow of information potentially crosses distribution boundaries, these decisions deserve extra attention in terms of their impact on concerns like performance, security, scalability, etc.

The Signature Element. This element further constrains the pointcut in terms of the signatures of the methods that should be intercepted. A signature is a *logic sentence* that consists of values combined with the three basic logical operators: and, or, and not; or conjunction, disjunction and negation, or complement. The values in a sentence are *method patterns* or *property interfaces*. A method pattern is an expression of the form:

```
"returnType methodName ( [parType1 [, parType2, ...]] )"
```

In this expression, `returnType`, `methodName` and `parType` refer to the names of return datatypes, methods and parameter datatypes, respectively. `returnType`, `methodName` and `parType` can also be replaced by the asterisk wildcard character "$*$", indicating that anything will match that particular name in the pattern. A method pattern will resolve to $True$ if the expression matches the actual method that is considered a join point for interception.

A value can also be a property interface, in which case it resolves to $True$ if the actual method under consideration is a member of that property interface. Technically, using a property interface in a pointcut comes down to specifying a signature that is a disjunction of all the methods in that property interface. However, this solution does not have the advantages of reusability and stability

that the property interface enjoys. Both method patterns and property interfaces can be used alongside each other in one logic sentence.

An example of the signature element is shown in Listing 2.12 (line 9), where the signature is defined to be the `@ServiceUsage` property interface. Without the definition of `@ServiceUsage` in Listing 2.2, this pointcut signature would otherwise have been specified as a disjunction of its four methods:

```
signature:
  ContentItem fetchArticle (ContentItemId) or
        List listNewestArticles (int) or
        List listArticlesForTag (Tag) or
        List listArticlesForCategory (Cat);
```

The Actor Element. The actor element constrains the pointcut further in terms of the following join point context property types: the *interface*, the *component*, the *host*, the *instance* and the *application*. All the properties are optional. Each property accepts a comma-separated set of values from its respective type, that are combined using disjunction. This means that, for a particular property, the join point under consideration must match (at least) one of the values defined for that property. It supports negation, but no wildcards. The properties are defined as follows:

interface This property takes a set of interfaces. At least one interface must contain a method that matches the signature of the join point under consideration. In case of a join point with call semantics the interface plays a required role, with execution semantics it plays a provided role. In case of negation, the method under consideration may not be a member of the negated interface.

component This property takes a set of component types. One of these types must match the type of the component that calls or executes the method of the join point under consideration. In case of negation, the method under consideration may not be a called or executed by the negated component.

host This property takes a set of host names. One of these host names must match the host where the component, that calls or executes the join point under consideration, is allocated. In case of negation, the component of the join point under consideration may not be allocated on a host with the negated host name.

instance This property takes a set of component instances. One of these instances must match the instance of the component that calls or executes the join point under consideration. In case of negation, the component instance of the join point under consideration may not match the negated component instance.

application This property takes a set of applications. One of these applications must contain the component that calls or executes the join point under consideration. In case of negation, the application containing the component of the join point under consideration may not match the negated application.

Listing 2.12 shows an example of a callee actor element with constraints for three properties, the interface, the component, and the host. The interface property is constrained to the `NewBrowse` interface, the component property to the `NewspaperService` component, and the host to either `PublishingServerA` or `PublishingServerB`.

2.2.6.2 The Selection of an Advice Method

An advice is a structured expression that defines which service should be executed, when that service should be executed relative to the intercepted join point. In MViewADL, the advice can be expressed in terms of the method that acts as the advice, in terms of the type of advice (`before`, `after`, and `around`), and optionally on which specific component instance the advice method should be called.

Any method of any interface can be selected to act as the advice of an AO-composition. When an advice method is selected, the connector must define a require-dependency on the interface that contains this method. For example, the AO-composition in Listing 2.12 has an advice of type *after* that selects `chargeForUsage(..)` as the method to act as its advice. This method is defined in the `Accounting` interface, which is declared as a dependency of this connector by means of its *require* definition on line 2.

Revisiting the 'Composition of Components' Property. Here we explain the third and last property of the AO-composition in MViewADL. A connector that captures an AO-composition consists of an advice and a pointcut. The dependency introduced by an advice is no different from a require interface in an OO-composition. However, this is not so for the pointcut, which represents a

very dynamic dependency that is not explicitly modelled as one through the static require part.

Still, the pointcut remains concerned with describing join points in terms of the signatures of methods. These methods are part of interfaces, that can only be provided by components. Because both the pointcut and the advice end up depending on components, an AO-composition is, like the OO-composition, concerned with the *composition of components*.

Figure 2.6 shows the `NewspaperAccountingCn` connector *composing* the `Accounting-Service` and `NewspaperService` components using an aspect-oriented composition. The figure models the OO-dependency of the connector on the `Accounting` interface that provides the advice method. In addition, it models the AO-dependency of the connector on the join points in the `NewsBrowse` interface of the `NewspaperService` component. This is a model of the AO-composition in terms of the Module view, which cannot capture all the details in the pointcut (like instances, hosts and applications).

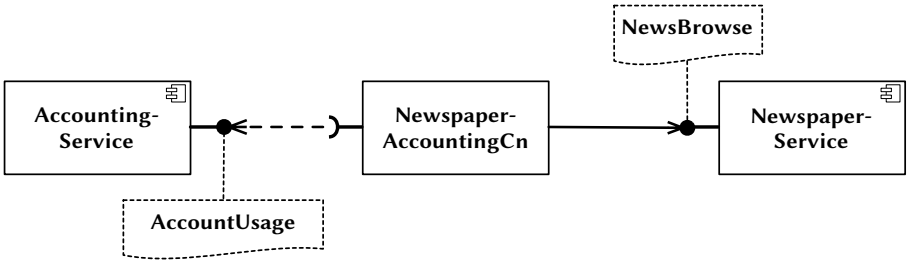


Figure 2.6: The graphical representation of the AO connector lacks the details of its MViewADL representation

2.2.6.3 Ambiguity in AO-Compositions

Like with OO-composition, the require dependency on the interface that contains the advice method, suffers from composition ambiguity when more than one component provides this interface. We will address this problem when we discuss module instantiation in Section 2.2.7, Configurations.

2.2.7 Configurations

The elements that have been introduced up until now, the interface, the component, the connector, etc. are used to describe the individual building blocks of a system. To build a distributed system, an architect combines these building blocks into a *configuration* and includes information on the distributed nature of the system by defining where these building blocks are to be *allocated*. This process is called *Application Assembly*.

While the software architecture community has already reached a consensus on the representation of the interface and the component, and steps [BB05, Gar98, Kel07, Sha94] are taken to do the same for the connector, there is little agreement on how to best describe application assembly. There exist many languages that describe application assembly in their own specific way, from (1) architecture description languages to (2) deployment descriptors in middleware platforms.

Application Assembly in ADLs. Most ADLs that focus on describing the structure of a system do this in terms of high-level concepts and concerns such as components, their interfaces and dependencies, and, modularization, encapsulation, and coupling. In addition, some ADLs describe a configuration of component and connector instances, while managing concerns such as availability and performance. Very few languages support the description of an abstract host topology and the allocation of instances onto hosts. In summary, the problem with current state-of-the-art ADLs is that they fall short in describing some of the concepts and concerns that are important to distributed system.

Assembly in Middleware platforms. In the case of deployment descriptors in middleware platforms, there is a practical need to come up with solutions for the application assembly and host allocation problems. Otherwise an application just will not get deployed. Here, deployment descriptors are used to configure the underlying middleware for hosting the particular application. They allow developers to identify components and specify their dependencies, to create an application by assembling components, and to deploy this application into a distributed environment. However, deployment descriptors are often only a part of the solution.

Application assembly in many middleware platforms is a combination of deployment descriptors and a development style that is implied, in part, by the selected component architecture (e.g. Enterprise JavaBeans) or middleware. Deployment

descriptors, or their dual, code-level annotation, are used to identify components and to specify their dependencies and various middleware-specific configurations. The development style, on the other hand, prescribes rules for the creation, structuring and the packaging of source code. Developers adhere to these rules in scripts that build, package, and deploy the application in accordance with some existing specification.

This way of describing is very pragmatic and a lot of the information (*configuration* and *allocation*) is hidden in processes, in the way the middleware works, and in various artefacts and code structures. In addition, because of the reasons laid out in the previous paragraph, deployment descriptors are irrevocably coupled to the component architecture or even the middleware that they are created for. As a consequence, architects run the risk of building architectures that are very hard to realize on top of other component models or even alternative middleware platforms.

Configurations in the Life-cycle. One reason for the apparent lack of consensus on the topic of application assembly is the complexity and variation that is involved in this process. Many industry-standard application middleware frameworks, like JBoss AS and the Spring Framework, are in a situation as described above. They do not yet offer an integrated way to deploy applications into a distributed environment. As a consequence, there is no way to declaratively describe configuration or allocation.

The research into distributed systems and middleware has shown the advantages of mastering and hiding distribution from the users of these systems. Recent research into middleware platforms such as JAC, AWED and DyMAC [LJ06, NSV+06, PSD+04], however, shows that having some controlled access to the distributed nature of an application, more specifically in composition and deployment, does have its uses. DyMAC even goes further by being the only middleware with deployment descriptors that support application assembly and deployment into a distributed environment.

The representation of a *Configuration* ranges from a configuration artefact in architecture to a combination of structured source code and scripts that build, package and deploy the application.

Because software architecture should describe the design of a system in a way that is most useful to the developers that make use of it, MViewADL was influenced

heavily by these recent advances in deployment descriptors. In MViewADL, application assembly is characterized by the following activities:

1. Specify a configuration that assembles component and connector instances.
2. Define a logical topology of hosts.
3. Specify the allocation of instances onto hosts.

2.2.7.1 The Application Element

The configuration and allocation of a distributed system in MViewADL is described using the *Application* element. The application element supports the description in terms of the *instances* of components and connectors (Section 2.2.7.2), and the *allocation* of instances onto an abstract and physical topology of *hosts* (Section 2.2.7.4). Figure 2.7 shows the instantiation of components on abstract hosts, in line with the `NewspaperApplication` configuration in Listing 2.13. In addition, the *application* also supports the in-line definition of modules —a grouping term for components and connectors. In-line definitions are preferred over global definitions if the module is specific to the particular application.

2.2.7.2 The Instance Element

It is common in a distributed system to have more than one instance of the same component. In MViewADL, a component is considered a *type definition*. It represents the properties of a component, rather then the runtime elements that are featured in the Component-and-connector view. For example (see Listing 2.13 and Figure 2.7), `NewspaperService` is a type of component, while `nsWeb` and `nsMob` (lines 11 and 12) are both instances of that component type that function independently at runtime. The same is true for the connector, which acts as the type definition for its instances.

There are various reasons for having more than one instance of a component. Each individual instance can be used to serve a distinctive client for reasons of security, performance, availability, etc. Instances can be used to divide up the workload of serving a large number of similar clients. Or, an instance can be used in another part of the system, for a different purpose (e.g. a staging newspaper service for production testing).

```
application NewspaperApplication {

  /*************** host definitions ***************/
  host PublishingServerA; host PublishingServerB;
  host WebServer; host MobileServer;
  host AccountingServer;
  host ContentManagementServer; host UserManagementServer;
  // ...

  /*************** instance definitions ************/
  NewspaperService               nsWeb     on PublishingServerA;
  NewspaperService               nsMob     on PublishingServerB;
  AccountingService              accs      on AccountingServer;
  WebService                     webs      on WebServer;
  MobileService                  mobs      on MobileServer;
  ContentManagementSystem        cms       on ContentManagementServer;
  UserManagementSystem           ums       on UserManagementServer;

  NewspaperAccountingCn          accnCn    on PublishingServer;
  ContentDeliveryCn(nsWeb,webs)  cdCn1     on WebServer;
  ContentDeliveryCn(nsMob,mobs)  cdCn2     on MobileServer;
  // ...

  /*************** component definitions ***************/
  component NewspaperService {
    provide { NewsBrowse, ConsumerProfile, NSManage }
    require { UserStore, ContentStore }
  }

  component AccountingService {
    provide { AccountUsage }
    require { UserStore }
  }

  component WebService {
    require { NewsBrowse, ConsumerProfile }
    provide { WebBrowse }
  }

  component MobileService {
    require { NewsBrowse, ConsumerProfile, Location }
    provide { MobileBrowse }
  }
  // ...

  /*************** connector definitions ***************/
  connector ContentDeliveryCn {
    require { NewsBrowse }
    provide { NewsBrowse }
  }

  connector NewspaperAccountingCn {
    // see Listing 2.12
  }
  // ...
}
```

Listing 2.13: The `NewspaperApplication` aggregates various architectural elements

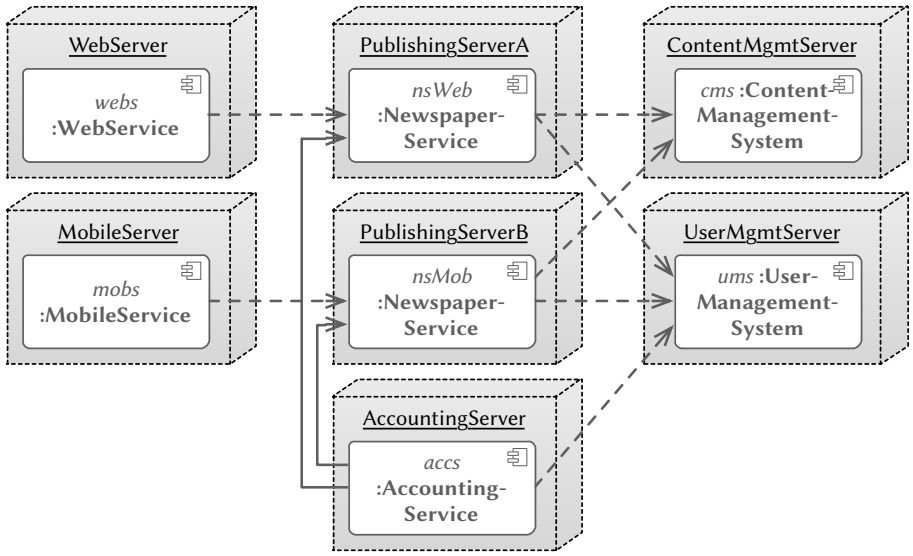


Figure 2.7: Allocation of component instances onto abstract hosts

Not surprisingly, the need for having connector instances is a consequence of supporting component instances. Defining multiple instances of a particular component type introduces the problem of composition ambiguity. Solving this ambiguity involves multiple parametrized instances of the connector that describes this composition. We discuss this in more detail in Section 2.2.7.5: *"Solving Composition Ambiguity with Parametrization"*.

In addition, instantiation is well suited to serve a number of other purposes in the ADL. We explain (1) how instance definitions relate to the explicit activity of application assembly; (2) why an instance definition is also responsible for the allocation of modules onto hosts; and (3) how connector parametrization solves composition ambiguity.

2.2.7.3 Application Assembly

An application description does not explicitly assemble component and connector types. Instead, it assembles instances of these modules that have either been defined in-line or globally. Furthermore, not every in-line component or connector definition may end up being used in the application under development.

Determining which modules are being assembled, is done simply by looking at the instances that are created in the application description. Components or connectors that have no instance, even if they are defined in-line, are not being used, and will not be part of the assembly.

2.2.7.4 The Host Element and Module Allocation

Host declarations are used to define the deployment topology of an application. Examples of hosts declarations are shown in Listing 2.13, lines 3–8. This topology is a logical one because its hosts are all abstract. The definition of logical topologies in a Component-and-connector view, allows architects to reason about the impact of distribution on the runtime structure of the application: *"Where are the inherent bottlenecks?"*, *"How do we structure our application to work around these bottlenecks?"*. Application-level solutions like load-balancing or caching will require a strategic positioning of instances and hosts.

In MViewADL, the instantiation of a module and the allocation of that module onto an abstract host is done with a single declaration. Every definition of an instance must declare the name of the abstract host where it is to be allocated. The reasons for this are practical: (1) *every* module instance needs to be allocated onto a host, and (2) the instance is the *only* element that can be allocated onto hosts. Examples of the allocation of an instance onto a host is shown in Listing 2.13, on lines 10–22. The component instance `accs` of component type `AccountingService` is to be allocated onto the `AccountingServer` host.

Application Deployment. Host declarations also support the definition of a concrete topology of physical hosts. Such a description is valuable for the application deployer that needs to deploy the system into a real environment.

To make a host concrete, its definition must be extended with a reference to the *Host Name* of a physical machine. This is done by appending the "`is`" operator and the host name string of that physical host. In Listing 2.14 the architect has adapted the application description with concrete alternatives for the host definitions. An graphical representation of this physical allocation excerpt is shown in Figure 2.8. The example illustrates that it is possible to map multiple abstract hosts onto a single physical host (*www.tomorrow.com*).

As long as an application description has logical hosts, it is considered abstract. An abstract application cannot be deployed into a physical environment.

```
application NewspaperApplication {
  // ...
  host WebServer                  is "www.tomorrow.com"
  host MobileServer               is "www.tomorrow.com"
  host PublishingServerA          is "news1.tomorrow.com";
  host PublishingServerB          is "news2.tomorrow.com";
  host ContentManagementServer    is "content.i.tomorrow.com";
  host UserManagementServer       is "users.i.tomorrow.com";
  host AccountingServer           is "bill.i.tomorrow.com";
  // ...
}
```

Listing 2.14: A fragment of the `NewspaperApplication` description that defines a physical topology

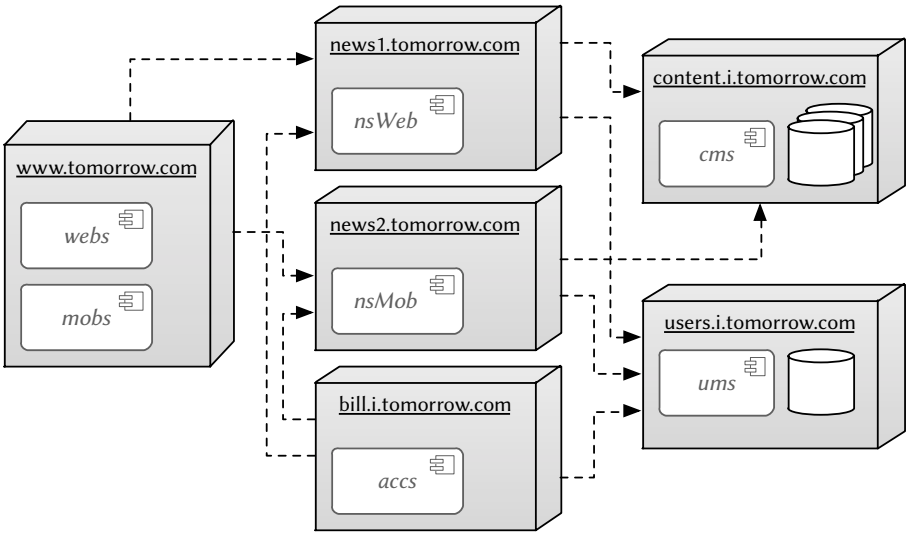


Figure 2.8: Allocation in a physical environment

2.2.7.5 Solving Composition Ambiguity with Parametrization

Allowing multiple instances of the same component, introduces the problem of *composition ambiguity*. The problem presents itself in various forms during development: module-level composition of different components providing the same interface, both in OO as AO-composition (Section 2.2.5.3 and 2.2.6.3) and, now, multiple component instances providing the same interface. The ambiguity

is a consequence of component composition that is based solely on matching a required interface to a provided interface with the same identity.

An example of an ambiguity is shown in Listing 2.13. The scenario in this example is one where the `NewspaperService` component has two instances `nsWeb` and `nsMob`, which indirectly serve web clients and mobile clients, respectively. The direct server-side clients of these newspaper instances are the `WebService` (line 35) and `MobileService` (line 40) components, respectively, that have one instance each: `webs` and `mobs`. However, both components depend on the `NewsBrowse` interface[2], that is now provided by the two component instances `nsWeb` and `nsMob`. This definition is ambiguous because it is impossible to determine which client instances (`webs`, `mobs`) are composed with which server instances (`nsWeb`, `nsMob`) based on interface matching alone.

Solving this composition ambiguity in MViewADL is done by *parametrizing the connector* in its instance definition with the instances of the components that it composes. An example of parametrization is shown in Listing 2.13 on lines 20–21. An instantiation requires two arguments. The first argument is the instance of a component that has the providing role in the composition, while the second argument is an instance that has the requiring role. Here, the `ContentDeliveryCn` connector is responsible for composing the instances of components that provide the `NewsBrowse` interface with those that require it, as specified in its definition (see line 47). This connector is instantiated two times, for the composition of `nsWeb` with `webs`, and `nsMob` with `mobs`.

Connector instance parametrization works the same way in the case of AO-composition ambiguity when there are multiple component (instances) that provide the interface containing the advice method. However, here the connector instantiation requires only a single argument, that of the providing component instance.

2.3 Validation and Discussion

Our validation of MViewADL consists of an application of the language in an industry-strength case-study, and of tool support that turns MViewADL models into source code for middleware platforms.

[2] We ignore the remaining require interfaces to simplify the explanation. Each dependency between interfaces requires its own connector definition, if it is ambiguous.

Experience with the language. One of the goals of the e-Media case study was to experience the development of a non-trivial application using state-of-the-art techniques across a big part of the development life-cycle [VOTV10]. As a result, we can make use of a large set of development artefacts ranging from requirements and an architecture, to a working and deployable implementation. This provided us with a lot of material to not only come up with an worthwhile problem statement, but also to experiment with our solutions by refactoring or replacing existing architecture artefacts with MViewADL descriptions.

An obvious thread to validity is that this validation is not empiric but rather illustrative. However, by using MViewADL in a non-toy case study, we gained experience with the language in a complex setting. In addition, in the process of getting our work published, we received invaluable feedback from reviews that allowed us to further refine the language, until it was finally accepted:

> *"Semantic integration of multiple views is one of the really big problems in software architecture. This [work] proposes a concrete and practical solution to this problem, with a well thought out language design, a prototype implementation, and a case study illustrating how the system works in a realistic problem setting"* —a reviewer

Input for code generation. Where we booked more tangible results on the effectiveness of our language is in the tool support that we provide. More precisely, in the fact that our code generation tools allow us to substitute existing code artefacts (in the implementation) by those that were generated from MViewADL descriptions. Running the application in a number of scenarios at least demonstrates the validity of the language and the architectural descriptions that we created. We revisit this code generation topic in Chapter 4 where we discuss multi-platform support and code generation.

Discussion

In this section we discuss how MViewADL fares in attaining the two goals put forward in this chapter: (1) Concepts of Distributed Systems, and (2) Integrating the Heterogeneous Expertise of Architects. We structure this discussion by going over each core element individually.

Interface. The interface is the enabler of loosely coupled interaction between components. An interface acts as a contract that grants designers and implementers of components and connectors a certain degree of independent development. The definition of an interface is not directly concerned with matters of distribution. Yet, the independence it introduces is an integral part of the management and operation of distributed systems.

The description of interfaces is part of the job of a module developer in the context of the Module view. This particular architect designs and describes the interfaces that will later be used in the definition of components and connectors. In the Component-and-connector view, interfaces will often appear in name only, as they represent the dependencies of components.

Component. The component is often called the smallest unit of independent deployment. The definition of a component in MViewADL, however, does not include descriptions concerning its allocation. Well designed components are loosely bound, by virtue of the contract that its interfaces define. This greatly simplifies their independent design, implementation and deployment.

The description of components is the job of a module developer in the context of the Module view. In addition, the component plays a role in the Component-and-connector view and the Allocation view. The Component-and-connector view deals with the instantiation of components, while the Allocation view determines onto which physical hosts these component instances will be allocated.

Connector. The connector is responsible for composing loosely coupled components. The connector supports adaptation of heterogeneous components with mismatching interfaces. Unlike the interface and the component, a connector that represents AO-composition is concerned with describing the distributed nature of the application. More specifically, the pointcut of a connector can express conditions in terms of the allocation context of components.

The description of a connector that captures an OO-composition is the job of a module developer in the context of the Module view. Such a connector is fully specified in terms of dependencies on interfaces —a module-level concern. The description of a connector in case of an AO-composition involves the expertise of many different architects in the context of various views. Such an AO-composition may specify conditions in terms of the Module view: *interfaces* and *components*, the Component-and-connector view: *instances* and *applications*, and

the Allocation view: *hosts*. The connector itself plays a role in the component-and-connector and Allocation view during the description of a *Configuration*, when it is instantiated and allocated onto a host.

Configuration. The description of an *Application* consists of four parts, the definition of hosts, the definition of instances, the allocation of instances onto host, and the (optional) in-line definition of modules. The impact of distribution on the *Application* is obvious and can be seen in the definition of a host topology and the allocation of instances onto hosts.

The definition of logical topologies of abstract hosts, the definition of module instances, and the allocation of instances onto logical hosts, is the job of an application assembler in the context of a Component-and-connector view. Extending the logical topology into a physical one is the job of a deployer, in the context of an Allocation view.

2.4 Conclusion and Future Work

MViewADL is an architecture description language with a focus on distributed middleware systems. The language is designed around core architectural concepts such as the interface, component, connector, and configuration.

The focus on distribution means that the ADL values loosely coupled components, substitutability of components, connectors that can adapt between mismatching interfaces, instantiation of components and allocation of instances on distributed hosts. It supports expressive composition, based on aspect-oriented techniques, that can access the distribution context of components.

MViewADL acknowledges that software architects reason about distributed systems in terms of multiple views. It supports aspects of the three common views when developing distributed systems, the Module, Component-and-connector, and Allocation view.

We validate the language through a combination of thorough experience in an industry-grade case-study and tool support that offers tangible results about the effectiveness of the language at the level of implementation.

MViewADL integrates the three views associated with distributed systems into a single consolidated language. This means that the concept of a view itself is

not a part of the language. It also means that it does not directly mirror *all* the knowledge that can be represented in these views (e.g. composited modules, specifications concerning performance and concurrency, etc.). It is considered part of future work to investigate these concerns in the context of our language.

Other limitations include the lack of support for specifying more complex application deployment, and a lack of support for specifying the behaviour of the architecture. Currently, application deployment is limited to the definition of hosts and the allocation of instances to hosts. Deployment in the context of cloud computing or larger infrastructures is much more complex than the e-Media case study. The current focus on structural concerns was intentional. It helped to keep the language simple. However, behavioural aspects of an architecture are too important to be ignored. However, this does not mean that we imply that a single language needs to tackle both structure and behaviour.

CHAPTER 3

Improved Modularity in Architecture Description

"Having divided to conquer, we must reunite to rule."

— Michael Jackson, ICSE, 1990 [Jac90].

The software architecture of a distributed system, like any large-scale software system, is not created in one day. It is developed over many iterations, across levels of abstraction, and by various architects, each with a particular role or expertise. An important part of developing software architecture is documenting that architecture. In this chapter we investigate the impact of the iterative process on the architecture description. More specifically, we look at the problems of current architecture description languages under iterative development, and we propose a solution that addresses these problems.

3.1 Introduction

We begin this chapter with a description of its context: iterative architectural development through individual changes. Next, we provide the problem statement and illustrate it by means of two examples. We follow this with an overview of our proposed solution. It consists of three parts: a concept, a technique, and its application in MViewADL. Each part of our solution will be addressed in detail in the three sections that follow (3.2, 3.4, 3.5). We conclude the chapter with an evaluation of our technique in terms of variability and modularity (3.6).

3.1.1 Context

A distributed system, like any large-scale software system, faces a continuous impact of change. *Software evolution* [BR00] is one cause of change that is generally associated with the long-term effects. In the short term, however, the iteratively developed software architecture of a distributed system faces change as well as.

Iterative development of the architecture means that its description is created in an iterative manner. As new requirements are considered, postponed design decisions are reconsidered, and mistakes are corrected, architects move the development of the architecture forward by contributing to its description. As our work focusses on the structural description of a software architecture using ADLs, architect contributions change the description by means of the addition, removal or modification of elements such as interfaces, components, connectors and configurations.

Research into software maintenance [LS80, BR00, Som10], and software architecture changes [WC07], has characterized[1] the changes to a software architecture into four distinctive categories: *Perfective*, *Corrective*, *Adaptive* and *Preventive*.

- *Perfective changes* result from considering additional requirements and from addressing deferred design decisions as development advances across levels of abstraction and across different views. An example of a perfective change is an extension of the existing `NewspaperService` component, to address the

[1] While these works operate from the perspective of the evolution of an initial version of the software system, we believe their characterization is still useful for the changes that occur in a more iterative development process, like for example Twin Peaks [Nus01].

requirement that clients of the newspaper service need to be able to *manage their user profile* (personal information, viewing history, payment data, etc.).

- *Adaptive changes* result from changes to existing requirements and from changes in the environment. For instance, a set of requirements that is modified to include an initial authentication step; a change to the requirements that concern deployment, to allocate into a new environment; and the re-evaluation of choices and trade-offs that are no longer optimal in that new environment. In the context of distributed systems, the allocation description is particularly relevant and very susceptible to change.
- *Corrective changes* result from correcting mistakes in the architecture that have been made in previous iterations. In addition to unavoidable human errors, sometimes trade-offs are made that conflict with later decisions. The architecture will have to be corrected in order to better satisfy the requirements. A corrective change is difficult to illustrate with a typical example. Basically, every description that fails to properly capture the intended solution is subject to correction.
- *Preventive changes* result from reasoning about what kinds of changes are likely to affect the software architecture in the system's lifetime. It is about the steps that are taken in order to minimize the impact of these future changes. An example of perfective changes is modifications that improve the variability of a software architecture.

As briefly mentioned when explaining perfective changes, architecture development is not only triggered by additional or modified requirements. In practice, a software architecture is developed (a) across levels of abstraction and (b) in terms of multiple architectural views[2].

a. Abstraction is instrumental in handling complexity and dealing with a (temporary) lack of knowledge. Abstraction is used to hide certain details, to allow for clearer communication, or to hide the fact that some details have not been worked out. For instance, an architect may consider the composition of a number of components, based on an initial understanding of their function, without having considered all the requirements that will contribute features to the components. Abstraction isn't a choice, it's a necessity.

[2]While architectural views are considered levels of abstraction themselves —some concern components and interfaces, other concern hosts and allocation— views are not the most fine grained interpretation of level of abstraction.

b. Architectural views capture the concerns of architects on specific aspects of the system, i.e. component interfaces, component composition, component allocation, etc. [ISO10]. Architects tend to focus on only one view at a time, while making abstraction of the knowledge contained in other views. As such, a view often coincides with the expertise of the architects that employ it. An elaborate software architecture considers the design of a system from a range of views. For example, development of distributed systems includes the views of *Module*, *Component-and-Connector* and *Allocation* [BCK13].

A system of iterative architectural changes, together with the concepts of abstraction and architectural view, support and describe the notion of iterative development of software architectures. It enables an architect to contribute to the description while making abstraction of matters beyond her expertise and beyond the scope of the contribution. She is able to focus on providing clients with access to news articles when defining the `NewspaperService` component, while making abstraction of profile management, component allocation, etc.

3.1.2 Problem Statement and Illustration

While the development of an architecture is widely considered to be an iterative process, the impact of iterative development on the description of the architecture has received less attention.

Problem

An iterative development process results in architects revisiting particular elements in the description and making changes to existing architectural descriptions. This issue is particularly relevant in the case of perfective and adaptive changes, but less so for corrective changes. For instance, to address the requirement concerning profile management, an architect may come up with a solution that requires him to revisit and change the `NewspaperService` component definition. An effect that is only further magnified by describing the architecture in terms of multiple views, as this encourages architects to collaborate on the description from different perspectives: existing elements are revisited in light of other views, by architects with a different expertise.

The most straightforward way, and often the only available way, of integrating changes, is by modifying the definitions of the elements in-line. However, this

results in *monolithic* definitions where the contributions of multiple architects are tangled. In other words, we are looking for a way to *modularize architect contributions* to the definitions of elements in architecture description.

The state-of-the-art in ADLs for software systems does not support a modular way to capture the changes to *all* elements across both (a) the levels of abstraction and (b) the various architectural views. This lack of modularity results in descriptions that suffer from a number of problems:

1. **The Loss of Traceability to Requirements and People.** The software architecture of a system is built from requirements. Architects develop solutions to address these requirements in the architecture. This process involves trading off both alternative solutions and conflicting requirements. This trade-off process results in a rationale that justifies, explains and documents the choices that led to that specific solution. Traceability is the ability to link requirements, solutions, and rationale together.

   Solutions are also linked to the person(s) responsible for it. When planning development there is the practice of assigning components or subsystems to teams. Designing the software architecture of a large-scale system is also a team effort that requires assigning responsibility. If an artefact of one architect is subsequently modified by another architect, it is no longer clear who is responsible for this artefact (e.g.: the `NewspaperService` component that is modified to include profile management). How do you come to an agreement on the owner of a model in case the knowledge involved exceeds the expertise of a single architect (e.g.: a configuration that is extended with a description of the allocation into a particular environment)?

   Achieving traceability to requirements and architects is difficult in a monolithic description because many of the solutions are tangled. To reduce tangling, we must improve the modularity of the description.

2. **An Impediment of Architectural Variability.** An architectural solution to a particular requirement is often just one of many possible alternative solutions. This is called variability in software development. While variability is often associated with software product lines [vGBS01], it is a key in the development of software systems in general [CABA09, Hil10, GA11].

   For instance, take the example of the user profile management solution. Adding profile management to the NewspaperService component, when using monolithic definitions, means that profile management becomes an integral part of the component. Combined with the loss of traceability, it

will be harder to come back on this decision later, and solve it in a different way. The ability of an architect to replace one architectural solution with an alternative solution depends on how well variability is managed within a software architecture [BB01].

Another interesting cause of variability in a software architecture is the environment into which the system will be allocated. This kind of variability has a clear impact on the usefulness of the software system as a product that can be deployed at the sites of multiple clients. A monolithic description of the architecture that has the deployment environment hard-coded, will be much harder to deploy into a different environment.

Architectural variability builds on the availability of traceability and modularity. Modularity enables reasoning about alternative solutions in isolation from the rest of the system, and traceability enables linking related alternative solutions.

3. **Lacking in Reuse and Abstraction.**

   Modularity, whether in descriptions or in code, is one of the characteristics that results in descriptions that are more reusable. Reducing tangling by modularizing descriptions increases the opportunity for reuse. Describing the `NewspaperService` component and its profile management extension separately, simplifies the creation of additional extensions to `NewspaperService`. Reusing the bulk of the architecture description for a deployment in a different environment, is another example of reuse. As such, variability itself depends heavily on the underlying mechanism for artefact reuse.

   Abstraction is the key to handling the lack of knowledge during design. This means that the design process will produce partial models that do no yet represent a complete system. Without support for abstraction in description languages, these partial models may often be invalid when held against the syntax and semantics of the description language.

Illustration

We illustrate the modularity problem in two examples in terms of two ADLs. The first illustration is a component definition in MViewADL that includes two tangled architect contributions. The second illustration is a connector definition in FractalADL where we show the contributions of three architects in the description of an AO-composition.

```
component NewspaperService {
   provide {
       NewsBrowse,
       NSManage
       ConsumerProfile,
   }
   require {
       ContentStore,
       UserStore
   }
}
```

Listing 3.1: The `NewspaperService` component in MViewADL, combining two architect contributions into a single description

A Tangled Component in MViewADL. The illustration in Listing 3.1 shows the `NewspaperService` component definition in MViewADL. What might seem a trivial component definition, is already the result of at least two architect contributions. The interface dependencies that belong to different contributions are highlighted by a different shade of grey. Those marked in lighter grey trace back to the requirements concerning the *"delivery and management of news content"*, while the others are concerned with the *"consultation and management of customer profile data by the user"*.

This simple example illustrates that once multiple contributions are applied to a single element definition, tracing each one back to the right concern, requires careful consideration and detailed knowledge of the newspaper service. In addition, once the profiling solution is applied, the simple `NewspaperService` component ceases to exist as an alternative solution. To solve these problems, we would need a way to capture both contributions to the `NewspaperService` component separately.

A Monolithic Connector in FractalADL. The illustration in Listing 3.2 shows a small subset of the NewspaperApp description in FractalADL [PSDC08]. We use an ADL from the related work to show that the modularity problem is not limited to MViewADL. The NewspaperApp description consists of two component definitions and a connector definition. The two components are `AccountingService` and `NewspaperService`. In this example, they each define one provided interface. The `server` role denotes a provided interface. The connector represents an AO-composition and consists of a binding and a weave, denoted by the `binding` and `weave` elements. In the weave, the `root` attribute indicates the scope to which

```
<definition name="NewspaperApp">
   <component name="AccountingService">
      <interface name="Accounting" role="server"
           signature="org.objectweb.....AspectComponent"/>
      ...
   </component>

   <component> name="NewspaperService">
      <interface name="NewsBrowse" role="server"
           signature="Service"/>
      ...
   </component>

   <binding
      client="NewspaperService.NewsBrowse"
      server="AccountingService.Accounting"
   />

   <weave root="this"
      acName="AccountingService"
      aDomain="ServiceAccounting"
      pointcutExp="
         <!– PublishingServer; (unsupported description) –>
         SERVER *;
         NewsBrowse;
         fetchArticle(ContentItemId):ContentItem"
      />
</definition>
```

Listing 3.2: The `ServiceAccounting` AO-Composition in FractalADL, combining contributions from three expert architects into a single description

the AO-composition applies. The `acName` attribute indicates the component that supplies the additional behaviour (advice). The `pointcutExp` attribute contains the *conditions* for composition, and must conform to the following template:

```
"(client | server | both) component; interface; method:type"
```

The `client` and `server` keywords denote an expected outgoing or incoming call respectively, followed by a `component` name, `interface` name, and `method` name, and return `type`. Names can make use of wildcards. The *pointcutExp* must be enclosed in quotes, individual fields must be separated with a semicolon and their order is fixed.

As indicated by the different grey areas, the `pointcutExp` element is a monolithic definition that is tangled with the contributions of multiple architects with a different expertise:

1. The first contribution from the bottom (line 25–27) is a Module-view description that focusses on interfaces and their methods. It describes a pointcut that matches calls of the `fetchArticle(..)` method, of the `NewsBrowse` interface. This is the responsibility of a module developer familiar with the newspaper service and the accounting requirements.
2. The next contribution (line 24) further specifies the pointcut in terms of the Component-and-connector view. Only incoming calls are allowed for any (indicated with *) component instance. This is the job of an application assembler.
3. The final contribution specifies that the called component is required to be allocated on the *PublishingServer* abstract host (line 23). However, since this last condition is not supported in Fractal-ADL, we had no choice other than to specify it informally.

This example illustrated how the views of different architects with respect to a single composition are tangled in a monolithic connector definition. Furthermore, it shows that the rigid structure of this particular FractalADL definition is difficult to develop iteratively.

3.1.3 The Multi-view Refinement Solution

We propose *Multi-view Refinement*, a *concept and technique* for the separation and modularization of the contributions of architects to the architecture description of a distributed software system. Our solution consists of the following three contributions, which are further described in the remaining sections of this chapter.

1 **Concept.** The concept of multi-view refinement introduces the idea of stepwise refinement of architecture description through the modularization of architect contributions. It enables architects to contribute to the description separately, at the appropriate time, and in terms of the concepts that match their expertise.

2 **Technique.** We define a generic technique, called ReVew, for the stepwise development of architecture description using ADLs. ReVew is an implementation of the multi-view refinement concept, that builds on the mechanism of inheritance in object orientation.

3 **Application.** ReVew, our generic technique, is applied to a heterogeneous set of non-trivial elements in MViewADL, and evaluated in terms of the effect on modularity and variability in an extended case-study of the e-Media system.

3.2 The Concept of Multi-view Refinement

In this section we explain the concept of multi-view refinement as it applies to the development of software architecture. We start with a look at the origin of the fundamental concept of stepwise refinement, and we explain what we mean with multi-view refinement, and how it is different from other stepwise refinement approaches. Next, we take apart multi-view refinement and define it as a set of requirements for ADL features. We end this section with a detailed look at the related work, where we analyse existing ADLs in terms of our proposed set of requirements.

3.2.1 Defining Refinement

Multi-view refinement is inspired by the fundamental concept of *stepwise refinement* in software development [Wir71], but applied to software architecture design. In this quote from 1971, Niklaus Wirth talks about a form of gradual development that he calls Stepwise Refinement:

> *The program is gradually developed in a sequence of refinement steps. [...] Every refinement step implies some design decisions. It is important that these decision be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. [...] A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible. This will result in programs which are easier to adapt to different environments.*

Wirth stresses the importance of design decisions in software development. In the context of his paper, Wirth uses the term *programming or program development*, as opposed to *coding*, to refer to the importance of design in software development. While the quote comes from a context of program development in the small, the basic idea still applies to how software architecture is practised today.

Stepwise refinement is defined as a software design approach that follows a scheme of top-down development across a series of steps. Each step involves taking part of the design (a system, a behaviour, a module, a method, an instruction, etc.) and decomposing it into a decreasingly abstract structure. This process is repeated

until the design reaches the desired level of detail. In other words, stepwise refinement is about taking a complete, yet abstract solution to a problem and refining it into a program that is straightforward to implement in a predetermined paradigm. The challenge lies in decomposing each abstract action into a set of actions that have to be completed in a particular order, until every action is trivial.

We define Multi-view Refinement conceptually as an approach to structuring architecture description in such a way that it enables architects to specify their contributions separately, at the appropriate time during development, and in terms of the architectural concepts that match their expertise. Like stepwise refinement this requires a process of iterative refinement of the design, that values weighing off alternative solutions, and that moves the design from an abstract state to one that has the desired level of detail. The challenge of Multi-view Refinement is decomposing the architectural description across multiple views and levels of abstraction.

3.2.2 Multi-view Refinement

Multi-view refinement enables the iterative development of a software architecture through a series of design steps, while insisting that these refinement steps are captured as separate, modular contributions that are part of the architecture description. In essence, it is about correcting the mismatch that exists between an architecture development process that is effectively iterative and the monolithic architecture description that this typically creates. The concept is inspired by the work on developer stakeholder roles [AF01, Szy02] and role-based task division of AO-composition [Lag09] in the context of distributed middleware systems.

Stepwise refinement has applications at many levels in software development. The term is most often used to indicate a development process that is highly systematic. A process that is essentially stepwise, in which each step deals with a specific, properly outlined modification that brings the design from one state to the next, while maintaining a set of properties. The principle of stepwise refinement is applied in many different fields, including formal modelling, design, and programming, but also model-driven architecture. We will illustrate this in our overview of the related work, later on in this section (Section 3.3).

The effects of stepwise refinement on the models and artefacts that are created by these approaches are varied as well. Some approaches modify models in-line as part of the process, while other approaches transform models from one format

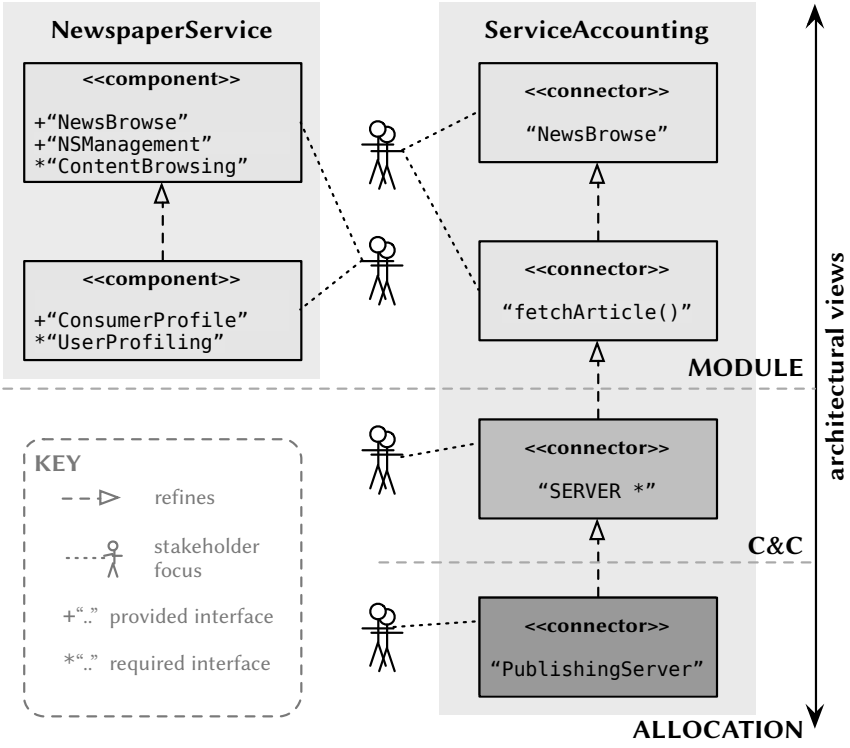


Figure 3.1: Multi-view refinement of the `NewspaperService` component (left) and the `ServiceAccounting` connector (right)

to another. Only few approaches use models that are aware of refinement and support it in their specification.

Multi-view refinement differs from the related work in that it employs stepwise refinement as a way to decompose and modularize architecture description in terms of the individual contributions of architects. Each refinement, the result of a careful consideration of the alternative solutions to a particular requirement or a postponed decision, is captured as an architect contribution.

In addition to being stepwise, multi-view refinement is also incremental. The architecture description is refined through non-overlapping contributions that must be considered in aggregation in order to see the whole picture. It allows a

particular contribution to focus on a particular task in terms of a particular view, without the need to reproduce or modify the descriptions of past tasks.

An illustration of multi-view refinement is shown in Figure 3.1. It revisits the two examples of the problem illustration section: the `NewspaperService` component on the left and the `ServiceAccounting` connector on the right. Mind that this illustration is not meant to be a visual alternative to refinement in the ADL. Its intention is to convey the concept of incremental refinement of architectural elements.

- In the example on the left, the component is shown consisting of two contributions that are related through refinement. The first contribution defines the basic newspaper service, while the second defines the addition of profile management to the newspaper service. The second contribution is defined incrementally, the definition of the component that is being refined, does not need to be replicated.
- In the example on the right, the connector is shown consisting of four contributions, in terms of the concept of three views. The first contribution defines the composition to intercept calls to the `NewsBrowse` interface, more specifically the `fetchArticle(..)` method, as defined in the second contribution. The third contribution further limits composition to incoming calls on a provided interface. Finally, the fourth contribution further limits composition to components allocated on the `PublisingServer` host. Again, refinement happens incrementally. Each contribution does not replace the definition of the composition, but adds to it.

Obviously, there exists some middle ground between capturing every single requirement or postponed decision as a separate contribution, and describing the entire architecture as a couple of monolithic models. For multi-view refinement to scale, it is important to decompose into contributions of sufficient size. For instance, adding profile management to the `NewspaperService` component should be at most one contribution instead of three —one for each additional interface dependency.

In principle, not every kind of architectural change should result in an architect contribution. Often the proper way to handle corrective changes is by in-line modification, as they are the result of correcting errors in the description. Perfective changes, on the other hand, often demand a non-trivial modification of one or more elements in the architecture description.

Using multi-view refinement allows incremental modification of an element definition, which has the benefit of preserving the element's original form and its traceability relation to the original requirement. The modification itself is modelled separately and can also be traced back to its own requirement.

In the following section we identify four requirements that define a multi-view refinement technique.

3.2.3 Requirements for a Multi-view Refinement Technique

It is our goal to enable architects to specify their contributions to the architecture description separately, at the appropriate time during development, and in terms of the concepts that match their expertise. Instead of a monolithic description, we expect an element definition to be modularized into multiple explicitly related definitions, that are no longer tangled with respect to the contributions of architects.

Further breaking down this goal allows us to propose three requirements for ADL features that we deem necessary in order to realise it:

1. Definitions Open to Adaptation
2. Multiple Models and Views
3. Model and View Relations

The requirements are inspired by the work on the Classification and Comparison Framework for ADLs by Medvidović, et al. [MT00]. We define and explain the requirements in detail, and illustrate with examples. Support by existing ADLs is classified into *no support*, *limited support*, and *full support*. We define what these categories mean in the context of each requirement.

1. **Definitions Open to Adaptation.** The iterative development of an architecture description is only possible if the ADL supports the detailed definition of the elements it provides (e.g. interface, component, connector, configuration, etc.), and, preferably, in a manner that is *transparent* and *open* to systematic adaptation. It should be possible for adaptation to be supported by a tool, or it should be part of the language. For instance,

models containing monolithic structures and overly complex models are harder to adapt systematically.

An opaque component definition does not support a detailed description of its meaning and cannot be iteratively developed. A connector definition that describes which components it composes and in what way, supports detailed description. An ADL has limited support if it has some elements that lack a detailed description or do not support systematic adaptation. An ADL has full support if its elements allow detailed descriptions (transparent) that can be systematically adapted. The requirement is supported by the *semantics* and *evolution* features in the comparison framework [MT00].

2. **Multiple Models and Views.** The description of an architecture consists of multiple models in terms of varying views. A single view may consists of many models that describe different (or overlapping) parts of the architecture from the perspective of that view. Each model consists of a set of (recurring) architectural elements that are related in some specific way. For instance, in the context of a module view, one model can capture the components and all their dependencies (connectors) in a particular subsystem of the architecture, while another model targets another sub-system, or even a specific interesting sub-set of components.

   An ADL has does not support this requirement if it only supports description in a single model. It has limited support, if it allows description by means of several models instead of just one. The ADL has full support, if it allows models that target description of the architecture by means of multiple models in terms of multiple architectural views. For instance, in addition to the previous models, a model of the component-and-connector view, that describes these components and their connectors from a runtime perspective (instances and composition in a runtime context). The requirement is supported by the *Multiple Views* feature [MT00].

3. **Model and View Relations.** All models are related in architecture description, but some relations elicit explicit description. This is the case for the models in the *Multiple Models and Views* requirement, where two models describe the same element from different contexts. To document the connection between these models, an explicit relation is required. One such relation is *unification*, which links identical elements in different models [Bou09]. However, more elaborate relations are possible, for instance one that defines the more abstract and the more concrete member in the connection.

While relations between models of the same view can be expressed in a more straightforward manner, relations between models belonging to different views are challenging because these views are often inconsistent [NKF03]. An ADL with relations between models of the same view has limited support for this requirement. Support for relations between models of different views, results in full support for this requirement.

We follow up this definition with a study of the support for these requirements in the related work (Section 3.3).

3.3 Related Work

We consider the related work of ReVew from two angles. From one angle we look at refinement and inheritance techniques in isolation (bottom up), while from a second angle we study the support of various ADLs for our multi-view refinement requirements (top-down).

3.3.1 Refinement and Inheritance

Stepwise refinement of software architectures is of course not new. Related work exists, by Moriconi, et al. [MQR95], Canal, et al. [CPT99], Baresi, et al. [BHT+04] and by Oquendo [Oqu04b], that considers refinement in the context of formal modelling. These approaches employ refinement as a sequence of steps that turns an abstract architecture specification into a concrete, implementation-oriented specification, with a focus on provable correctness.

Other related work considers a more practical approach to refinement of the software architecture from abstract to implementation. Denford, et al. [DL03] define refinement as going from an abstract to a more concrete model, that satisfies a larger subset of the requirements of the system. Abi-Antoun and Medvidović [AAM99] consider how UML models can assist in the refinement of a software architecture. While Batory, et al. [BSR03], Lopez-Herrejon, et al. [LHBL06], and Apel, et al. [AKLS07] consider stepwise refinement at the programming level. These works focus on on refinement of behaviour in programming modules, while ReVew is concerned with architectural structures. They discuss ways in which simple programs can be refined incrementally into programs that are more feature-complete.

Furthermore, we acknowledge a body of work on incremental extension in the context of object-orientated languages, but we consider this to be outside the scope of this work.

3.3.2 Analysis of Support in ADLs

In this section we analyse the support for the multi-view refinement requirements of Section 3.2.3 in a range of ADLs for distributed systems. Table 3.1 shows the features that are supported by the various languages in the related work. For each requirement we discuss the most important observations.

1. **Definitions Open to Adaptation.** ADLs a–m support detailed definitions for all or most of their elements. Although, the flexibility and complexity of these definitions varies widely (e.g. a connector using unification based on port IDs to one with aspect-oriented composition).

   However, Fractal ADL has some monolithic structures that are hard to adapt systematically. Wright, Rapide and LEDA (h–j) formally capture behaviour in the architecture description, and while addition and redefinition of behaviour is sometimes supported, the adaptation of existing behaviour generally is not.

   Rapide deals with types and events, and the main concepts of AADL (m) are threads, processes, data, etc. This sets these languages somewhat apart from the other, which typically model components, interfaces and connections in one form or another.

   Some ADLs n–q have certain opaque elements (e.g. connectors, such as pipe-filter, defined by means of an architectural style) with pre-defined semantics. *Stratified Frameworks* and *Viewpoints Framework* are rather abstract in nature. Consequently, they do not focus on the detailed description of many of their elements.

   MViewADL supports the detailed definition of all of its elements, and all of its descriptions can be systematically adapted.

2. **Multiple Models and Views.**

   Of languages a–e, only AspectLEDA and Prisma clearly demonstrate the description of an element by means of different models. The other languages support very hierarchical single-model descriptions that did not clearly illustrate or discuss the possibility of modularization.

Table 3.1: The feature matrix indicating support for our requirements in the related work

	Adaptive Definitions (1)	Multiple Models & Views (2)	Model & View Relations (3)
[PFT05] DAOP-ADL (a)	●	·	·
[PFT11] AO-ADL (b)	●	·	·
[PSDC08] Fractal-ADL (c)	○	·	·
[NPTM09] AspectLEDA (d)	●	○	·
[GCB+06] AspectualACME (e)	●	·	·
[PACR06] Prisma (f)	●	○	·
[Oqu04b] π-ADL^{ARL} (g)	●	○	○
[All97] Wright (h)	○	○	·
[Luc96] Rapide (i)	○	○	·
[CPT99] LEDA (j)	○	●	○
[MDK94] Darwin (k)	●	●	·
[MQR95] SADL (l)	●	○	○
[FLVC05] AADL (m)	●	●	○
[Bou09] View Composition (n)	○	●	○
[Gru00] Multi Perspective (o)	○	●	○
[AK03] Stratified Frameworks (p)	·	○	○
[NKF03] Viewpoints Framework (q)	·	●	○
MViewADL (r)	●	●	·

· – no support ○ – limited support ● – full support

Languages g–m all allow a modularization of the description into multiple models. Although some only earned a limit-support bullet because they have a flat description space, lacking a higher-level concept (*application*, *architecture*, *system*, *configuration*, etc.), that would complicate modularization (Rapide, Darwin). LEDA, Darwin and AADL all support module view descriptions of components, and the notion of component instantiation. Only AADL goes further, by also supporting the concept of allocation to physical elements (CPUs).

Of languages n–q, Stratified frameworks supports multiple models, but only in terms of a single view. Languages n, o, and q on the other hand, support multiple models and views.

MViewADL supports the description of an element by means of multiple models and supports models with concepts from various views (interfaces, components, connectors, applications).

3. **Model and View Relations.**

   Languages a–f do not offer model or view relations. Of languages g–m, π-ADL^{ARL} and SADL support a refinement relation between models, while LEDA and AADL support inheritance between particular elements. Some languages describe domain-specific relations between their elements. Rapide models dependency between events, Wright supports parametrisation of some elements —creating an implicit relation between abstract and concrete elements. We do not consider these relations to be suitable primitives for multi-view refinement.

   Languages n–q support relations between models, however, specific cross-view relations are considered feasible but challenging and are not detailed (Viewpoints Framework).

   MViewADL does not support relations between models, nor between models of different views.

Conclusion. The languages that we studied seemingly fit into one of three possible categories.

Languages a–f are part of the first category. They focus on describing components and how they are composed together using (aspect-oriented) connectors. They support open definitions, for the most part. These languages rarely support multiple independent models, let alone models belonging to different views. As a consequence, they are not concerned with capturing relations between models.

The second category includes languages g–m. These languages are concerned with describing, verifying and analysing the architecture of distributed, dynamic or embedded systems. It is a varied set of ADLs that is also apparent in varying degree of support for the requirements (Table 3.1). Nevertheless, overall this category has the highest degree of support for the requirements.

Languages n–q are part of the last category. These focus on capturing an architecture in terms of multiple models, often across multiple views, and the relationships between these models. Documenting cross-view relationships between models, however, remains challenging because of the general-purpose nature of these languages and the views they support [NKF03]. These languages often make abstraction of the details of some elements, like components or connectors.

Now that we have introduced the concept of multi-view refinement and studied the related work in detail, we will develop a technique that can be used in ADLs, in the next section.

3.4 ReVew — A Multi-view Refinement Technique

ReVew is a technique for the iterative development of architecture descriptions that implements the concept of multi-view refinement for the use in ADLs.

We define and describe ReVew in a generic way to avoid targeting a specific element or a particular ADL. Our technique builds on the concept of stepwise refinement and the mechanism of inheritance. ReVew takes on the form of a refinement relation between elements of the same kind, that manages the inheritance of the members from parent to child by means of *overriding* and *merging*. While inheritance supports override as an all-or-nothing way of redefining parent statements, merge allows for a much finer grained mechanism for redefinition. The merge operation is instrumental in realizing the collaboration of architects across multiple descriptions and in terms of various views.

We will now explain the abstract grammar and the semantics of ReVew. First we describe the abstract form of the *RefinableDeclaration* (Listing 3.3), and illustrate it on a small example. Finally we reduce refinement to its most basic operations, *override* and *merge*, and explain both in detail and illustrate with the example.

```
Declaration:
   RefinableDeclaration | FinalDeclaration

RefinableDeclaration:
   abstract? Keyword Id ( refines Id ( "," Id )* )? Body

Element:
   Declaration | ElementWithoutName | DeclarationReference

FinalDeclaration:
   Keyword Id Body?

ElementWithoutName:
   Keyword Body

DeclarationReference:
   Id

Body:
   "{" ( RefMod? Element ( LF | "," )? )* "}"

Id: "an alphanumeric value that identifies this declaration"
Keyword: "a keyword that indicates the type of element"
RefMod: override | merge
```

Listing 3.3: Abstract grammar for declaration refinement

3.4.1 Refinement of Declarations

Multi-view refinement of declarations is similar to the technique of inheritance in object-orientation, but it differs in the kinds of members that can be *inherited* (refinable declarations, final declarations and elements without a name), and in the way these members can be *redefined* (override and merge).

The term refinement has a rather broad meaning and application. Talking about refining a class, an interface, a component, a pointcut or even an architecture, makes sense intuitively. It triggers the thought that we are improving the particular element through small, precise changes. In a general sense, every element in a description can be refined. However, when we talk about the refinement concept here, not every element is subject to refinement. In the context of a language, the refinement technique that we propose only applies to declarations, i.e. language elements that have an identifier. A declaration must be able to refer by name to the declaration that it refines.

```
abstract component NSClient {
   require {
      NewsBrowse,
      ConsumerProfile
   }
   provide {
   }
}

```

Listing 3.4: The abstract component `NSClient` defines a generic client of the newspaper service

```
component MobileService refines
     NSClient {
   require {
      Location
   }
   provide {
      MobileBrowse
   }
}
```

Listing 3.5: The `MobileService` component refines `NSClient` and inherits its require interfaces

The refinement grammar is shown in Listing 3.3. The declaration that supports refinement in this grammar is named *RefinableDeclaration*, in contrast with *FinalDeclaration* which is the declaration that does not support refinement.

3.4.1.1 The Mandatory Parts

The mandatory parts of a refinable declaration are a declaration keyword (`Keyword`), an identifier (`Id`), and a body (`Body`). The declaration keyword indicates the type of the declaration: e.g. class, interface, component, etc. The identifier is the name of the declaration, i.e. the name of the class, interface, component, etc.

The body consists of *zero or more* language elements —also called its members— and is separated from following elements by a linefeed or a comma. An element is preceded by an optional *refinement modifier* (`RefMod`) that determines how the particular member is to be redefined. The possibilities for redefinition are *override* and *merge*. We explain the meaning of these modifiers further in Section 3.4.2 when we discuss the redefinition of members under refinement. It is up to the designer of the language to select a sane default modifier for each type of member of a refinable declaration.

Listing 3.4 shows the definition of a refinable declaration of type *component* with name `NSClient`. The component consists of two nameless elements: `require` and `provide`. While the `provide` element is empty, the `require` element has two members: `NewsBrowse` and `ConsumerProfile`. Both members are *declaration references* to interface declarations that are defined elsewhere.

3.4.1.2 The Optional Parts

The optional parts of a refinable declaration are an abstract modifier and a *refines* part.

The refines part is designated by the `refines` keyword. It indicates a refinement of one or several refinable declarations by means of a list of their identifiers. This means that a refinable declaration supports a form of multiple refinement. While the list of names in the *refines* part are designated by `Ids`[3], the use of fully qualified names that refer to refinable declarations is also supported. The refinement part is optional because it should not be mandatory for a refinable declaration to refine declarations, if only, to allow bootstrapping of a refinement chain.

Listing 3.5 shows the definition of a refinable declaration of type component, named `MobileService`. It refines the `NSClient` component by merging the require element of the child with that of the parent, and by merging the provide element of the child with the empty one of the parent. This results in a `MobileService` that has a require element consisting of `Location` and the inherited `NewsBrowse` and `ConsumerProfile` members, and a provide element consisting of the `MobileBrowse` member that it supplied itself.

Abstract. The abstract modifier is designated by the `abstract` keyword. It determines if the refinable declaration is allowed to be incomplete, i.e. allowed to be missing essential members. If a declaration contains other declarations that are abstract, it is considered abstract itself, and must be declared that way. Abstract declarations can be made concrete later on by means of refinement.

In the example in Listing 3.4, the `NSClient` component is declared abstract because it serves as a generic client component for the newspaper service.

3.4.1.3 Refinement Defined

The ReVew technique defines refinement of declarations as follows:

> **If declaration B refines declaration A,** then declaration B has a `refines` part, that refers to declaration A by name.
>
> The refinements that B can perform are:

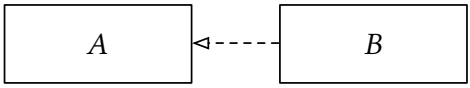
[3] To limit the size and complexity of the grammar.

1. Redefine members inherited from A by means of *merge* or *override*
2. Add new members, that are not present in A (including members that are missing, if A is abstract)

In addition, B inherits all members of A that are not redefined in the body of B. In this relationship A is the parent and B the child. Refinement is only supported between declarations of the same declaration type, and it is not allowed to create loops in the refinement graph.

Multiple refinement. Like multiple inheritance in object-orientation, multiple refinement enables a declaration to refine more than one parent declaration. Refinement will iterate through the set of parent declarations depth-first, from first to last, resolving each refinement relation individually and aggregating the results. Conflicts are detected using the *rule of dominance* [Str00]. This means that if a declaration inherits two different definitions for a member and they cannot be merged, then it must provide a new definition to resolve the conflict. Members that cannot be merged are: declarations and *elements without a name* that do not support merge. For multiple refinement to be usable in practice, we depend upon behavioural subtyping between refining declarations. There is currently no support to force proper usage of refinement. It is left to future work to further refine this aspect of ReVew.

Graphically, a declaration B refining a declaration A is represented as follows:



3.4.2 Redefinition of Members

The child in a refinement relation inherits all members of its parent, except for the ones that it redefines. The semantics of the redefinition depend on the kind of the member that is redefined.

3.4.2.1 Overriding of Members

When a child inherits a member that is a declaration and that already exists locally, redefinition always implies *override*.

Overriding an inherited declaration happens when the child in a refinement relation has a local declaration that has the same name as the declaration that is inherited. As a result, only the local declaration is accessible. On the other hand, overriding in the case of an *ElementWithoutName* only occurs when the element is annotated with the `override` modifier or the element does not support merge.

Because override is the only possible behaviour between declarations, emphasizing it by adding the `override` modifier to a declaration means that this declaration *should* have an inherited parent to override.

3.4.2.2 Merging of Members

When a child inherits a member that is not a declaration (element without a name) and that already exists locally, redefinition defaults to the *merge* technique.

Merging elements K and M is a structurally recursive operation:

1. First, it is verified whether the locally declared element has the `override` modifier, or if it is a declaration. This turns the *merge* behaviour into *override*, which results in choosing the locally declared element over the inherited elements.
2. Second, if it *does* allow a merge, all members of K will be merged, type by type, with members of element M, according to the merge semantics of that type.
3. Finally, this is repeated until an element is reached that does not have a body, that is a declaration, or that demands override semantics.

In Listing 3.5 the *require* of `MobileService` is merged with that of `NSClient`, resulting in a *require* of `NewsBrowse`, `ConsumerProfile`, and `Location`.

In the following section, we apply this generic technique to MViewADL.

3.5 Applying ReVew to MViewADL

Our application of the ReVew technique is its integration into MViewADL. MViewADL is our multi-view architecture description language for distributed systems that integrates the heterogeneous expertise of architects. The language supports the description of the architecture of the distributed system through the contributions of multiple architects in terms of concepts of the module, component-and-connector and allocation views.

When we described MViewADL in Chapter 2, we introduced the idea of a single architecture description language that integrates concepts from the three architectural views that are particularly suited for distributed system: the module view, component-and-connector view and allocation view. This allows architects to define architectural models that are more integrated, and as a consequence, more consistent than fully disconnected models. However, a disadvantage of the integrated model is that it combines the various contributions of architects into monolithic descriptions that suffer from the problems outlined in the problem statement in this chapter (see Section 3.1.2).

In this section, we explain and illustrate how ReVew applies, constructively, to a heterogeneous set of architectural elements in MViewADL, including the interface, the component, the connector, the AO-Composition, and the application.

In addition to extending MViewADL with ReVew, this section is also an illustration of the genericness of the technique itself. It is true that one application does not guarantee that ReVew can be integrated into related ADLs as successfully. However, its support for such a heterogeneous set of elements in MViewADL, that corresponds at least conceptually to the elements of related ADLs, lends some confidence in this regard.

3.5.1 The Interface

Refining an interface in MViewADL is the same as extending an interface using inheritance. Listing 3.6 shows interface `NewsAccess` refine the `NewsBrowse` interface that was defined in Section 2.2.2 (Chapter 2). The `NewsAccess` interface can be seen refining its parent with a new method, `searchArticles`, that allows to search for articles based on a set of parameters.

The only kind of member that an interface has, is the method. Because a method is a declaration, the only manner of redefinition that refinement supports is

```
interface NewsAccess refines NewsBrowse {
    List searchArticles(SearchParameters parameters);
}
```

Listing 3.6: Refining the `NewsBrowse` interface

```
component NewspaperService {
   provide {
      NewsBrowse
   }
   require {
      ContentStore
   }
}
```

Listing 3.7: A simplified `NewspaperService` component declaration

override. This means that every element inherited from `NewsBrowse` that has a local definition in `NewsAccess` with the same identity will be overridden. The identity of a method includes the types and order of its parameters.

3.5.2 The Component

Refining a component in MViewADL allows the extension of its provide and require members. Listing 3.7 shows the `NewspaperService` component declaration that represents a simplified newspaper service. This component provides the `NewsBrowse` interface to clients, and it has a dependency on the `ContentStore` interface to get access to content data.

The `NewspaperAccessService` component refines the `NewspaperService` component in Listing 3.8. While it inherits both the provide and require members from `NewspaperService`, it substitutes the provide element, that holds the `NewsBrowse` interface, with a provide element that holds the `NewsAccess` interface from Section 3.5.1.

When one component declaration refines another, it can redefine the members that it inherits. The *provide* element in a component is an *element without a name* that supports merge. However, here, we use the *override* refinement modifier to turn merge into override. This way, we replace all the inherited provide interfaces —in this case only one— with the provide interface listed here. In this case, we

```
component NewspaperAccessService refines NewspaperService {
   override provide {
      NewsAccess
   }
}
```

Listing 3.8: A refinement of the NewspaperService component that substitutes the provide interface

```
component ProfiledNewspaperService refines NewspaperService {
   provide {
      ConsumerProfile,
   }
   require {
      UserStore
   }
}
```

Listing 3.9: A refinement of the NewspaperService component with additional provide and require interfaces

end up substituting the `NewsBrowse` interface for `NewsAccess`. Because the refining component does not include a redefinition of the require element, it is inherited as is.

The `ProfiledNewspaperService` component in Listing 3.9 uses refinement to add support for customer profiles to the newspaper service. More concretely, it adds one provide interfaces: `ConsumerProfile`, and one require interface: `UserStore`.

In this refinement of a component, both the provide and require members of the parent component get redefined by the child. This time, we make use of the default merge redefinition of the these elements. This means that the provide of the parent is merged with that of the child, resulting in two provide interfaces for the `ProfiledNewspaperService` component. Similarly, merging the require elements results in two require interfaces for the `ProfiledNewspaperService` component. This result is shown in the `NewspaperService` component in Listing 2.3.

3.5.3 The Connector

Connector refinement in MViewADL, and by extension AO-composition refinement, allows the extension and modification of the AO-composition's pointcut

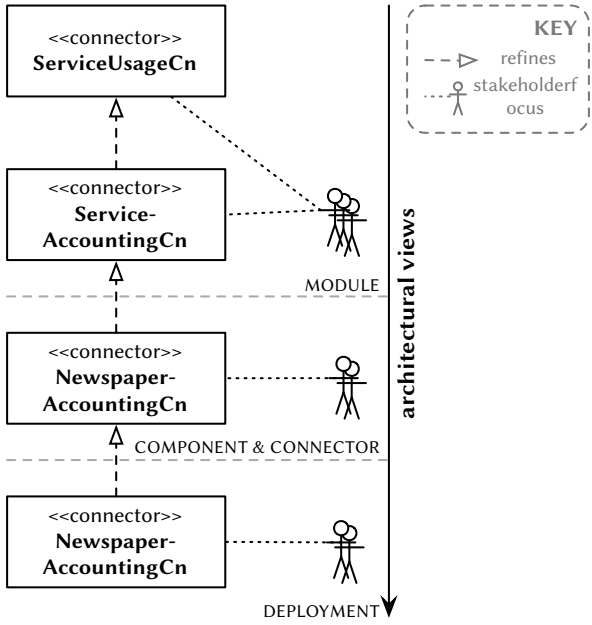


Figure 3.2: Multi-view refinement of an MViewADL connector

and advice, and the extension of the connector's require and provide members.

To explain connector refinement, we start the example with the `ServiceUsageCn` connector in Listing 3.10. This is the initial connector definition that will be refined across a number of steps, as illustrated in Figure 3.2. This connector has a single AO-composition member called `ServiceUsage`. The AO-composition is abstract because it is incompletely specified, as it contains only a pointcut element (lines 3–9) and no advice. The connector is abstract as well, because it contains a member that is abstract.

The pointcut in this AO-composition is of kind *execution* (line 4). It has a signature comprising the `@ServiceUsage` property (line 5). The pointcut is further constrained in terms of the callee context: the called component must provide the `NewsBrowse` interface (line 7).

```
1  abstract connector ServiceUsageCn {
2     abstract ao-composition ServiceUsage {
3        pointcut {
4           kind: execution;
5           signature: @ServiceUsage;
6           callee {
7              interface: NewsBrowse;
8           }
9        }
10    }
11 }
```

Listing 3.10: The abstract `ServiceUsageCn` connector and `ServiceUsage` AO-composition

3.5.3.1 Refinement of Declarations

Similar to the component, refining a connector in MViewADL allows the extension of its provide and require members. We will not repeat this explanation here, instead we refer to Section 3.5.2, where we discuss this in the context of component refinement.

Connector refinement becomes much more interesting when it involves AO-composition. While not mandatory, connector refinement often involves the refinement of one or more AO-composition declarations that it inherits from the parent connector, in addition to the refinement of its require and provide members. Listing 3.11 shows such an AO-composition refinement inside a connector refinement: the `ServiceAccountingCn` connector that refines `ServiceUsageCn`, and the `ServiceUsage` AO-composition that refines its parent by the same name. Refinement, however, does not preclude one AO-composition refining another AO-composition in the local context of a single connector, or more generally, one declaration refining another declaration in the same context.

The goal of this `ServiceAccountingCn` connector is to compose the Accounting service. This is done by refining the `ServiceUsageCn` connector and `ServiceUsage` AO-composition (Listing 3.10) that defines the reusable pointcut, and extending it with an after advice that calls the accounting service.

The refinement of the `ServiceUsage` AO-composition in Listing 3.11 (line 6) results in it inheriting all members of its parent (line 2 in Listing 3.10), without redefinition. In addition, this refinement adds an *advice* element to the AO-composition. The new advice element declares the *chargeForService()* method as

```
1  connector ServiceAccountingCn refines ServiceUsageCn {
2     require {
3        Accounting
4     }

6     ao-composition ServiceUsage refines ServiceUsageCn.ServiceUsage {
7        advice {
8           service: chargeForService();
9           type: after;
10       }
11    }
12 }
```

Listing 3.11: Refinement of a Connector and an AO-composition with an *advice* member

advice of type *after*. *chargeForService* is a method from the `Accounting` interface that plays the roll of advice for this composition. The AO-composition does not further refine the inherited pointcut element.

The `ServiceAccountingCn` connector (line 1) refines the `ServiceUsageCn` connector (Listing 3.10). The connector inherits the `ServiceUsage` AO-composition from its parent. However, because both the newly defined AO-composition and the inherited AO-composition share the same name, the inherited one is overridden. Because the `ServiceUsage` name now points to the one in the local context (line 6), it is necessary to supply the *fully qualified name* of the AO-composition that is being refined. In addition, the connector defines the *require* element (line 3). This dependency on the `Accounting` interface is required by the *advice* element in the `ServiceUsage` AO-composition, as `Accounting` provides the *chargeForService()* method.

Intermediate AO-composition Definitions. The fact that the name `ServiceUsage` is used consistently in each refinement of the AO-composition in listings 3.11, and 3.12 is intentional. The reason for this is that once a connector is instantiated, all of its non-abstract AO-compositions, inherited or defined locally, are *activated* within the application —i.e. their joinpoints are intercepted and advices executed. While this allows for a connector to consist of multiple AO-compositions with varying goals, this behaviour is undesirable when we are dealing with intermediate versions of the same AO-composition. By using the same name for each refinement, the override mechanism will prune the intermediate AO-compositions.

3.5.3.2 Redefinition of Members

```
1 connector NewspaperAccountingCn refines ServiceAccountingCn {
2    ao-composition ServiceUsage refines ServiceAccountingCn.ServiceUsage {
3       merge pointcut {
4          merge callee {
5             override component: NewspaperService;
6          }
7       }
8    }
9 }
```

Listing 3.12: The `NewsAccounting` connector refining the `ServiceAccounting` connector with a component condition

In Listing 3.12, the `NewspaperAccountingCn` connector refines the `ServiceAccounting-Cn` connector from the previous listing (Listing 3.11). The connector has one member: the `ServiceUsage` AO-composition. Because this AO-composition has the same name as the one that is inherited from the parent connector, the parent AO-composition is overridden. Again, because the `ServiceUsage` AO-composition is itself a refinement of the AO-composition that it overrides, the use of a *fully qualified name* (line 2) is necessary. Otherwise, it would create a loop in the refinement graph, which is not allowed.

The `ServiceUsage` AO-composition inherits the advice definition as is, and redefines the pointcut definition that it inherits from its parent. The pointcut is further constrained by limiting interception to the those callee components that are of type `NewspaperService`. In other words, only those join points remain where the callee is of component type `NewspaperService`.

When merging a pointcut:

1. The signature is merged with the signature of the parent by joining their lists of members through a disjunction. This is similar to an implicit disjunction with *super*, if both elements define a signature, and conforms to the joinpoint model of MViewADL (see Section 2.2.6.1). Because only the pointcut in the parent AO-composition has a signature definition in the example (Listing 3.10), this merge is trivial.
2. The *kind* of the child overrides the *kind* of the parent, if both elements define a kind. This is because the kind can only hold a single definition (either execution or call). This merge in the example is trivial too, because only the pointcut in the parent AO-composition has a kind definition (Listing 3.10).

3. The callee (and caller) elements are merged as well. Merging the callee (or caller) elements means merging each of the context properties of the same type: interface, component, instance, host, and application. Merging context properties is done using disjunction, i.e. aggregating both comma-separated lists into a single list. In the example, the pointcut has a callee member with a component property. However, because this property is marked with the `override` keyword (line 5), all prior definitions of the callee component property are ignored. We have explicitly added the superfluous `merge` modifier to the pointcut and callee elements to illustrate the contrast with the override of the component property.

When merging the advice, the *type*, *service*, and *instance* members of the child can only hold a single definition. As a consequence, the definition of the parent is overridden, if both child and parent have one. This is similar to the merger of the kind element in a pointcut.

In MViewADL, the elements that support merge are the require and provide dependency elements, the advice, the pointcut, its signature, the caller and callee, and their context properties: interface, component, host, instance and application. There are three elements that do not support merge, because they can hold only a single value: the pointcut member, kind, and the advice members, type and service.

3.5.4 The Application

Application refinement in MViewADL allows the extension of an application definition in terms of the hosts, the instances, and the module elements (components and connectors) that it defines.

For the explanation of application refinement, we start with an example that defines the initial `BasicNewspaperApp` application declaration (Listing 3.13). The purpose of this declaration is to define a basic newspaper application that can later be extended with additional services. `BasicNewspaperApp` declares a number of abstract hosts: `PublishingServer`, `ContentManagementServer`, etc. (line 2), and a number of instances: `cms`, a `ContentManagementSystem` instance that is allocated on the `ContentManagementServer` host, and `ns`, an instance of the `NewspaperService` component that is allocated on the `PublishingServer` host, etc. This application definition does not specify any inline module declarations.

```
1  abstract application BasicNewspaperApp {
2     host PublishingServer;
3     host ContentManagementServer;
4     host UserManagementServer;
5     host NewsDeskServer;
6     host Intranet;

8     ContentManagementSystem   cms   on ContentManagementServer;
9     NewspaperService          ns    on PublishingServer;
10    UserManagementSystem      ums   on UserManagementServer;
11    NewsDesk                  nd    on NewsDeskServer;
12 }
```

Listing 3.13: The `BasicNewspaperApp` application declaration

```
1  abstract application AccountedNewspaperApp refines BasicNewspaperApp {
2     host AccountingServer;

4     AccountingService      accServs  on AccountingServer;
5     NewspaperAccountingCn  accConn   on PublishingServer;

7     // connector from Listing 3.12
8     connector NewspaperAccountingCn refines ServiceAccountingCn {
9     ao-composition ServiceUsage refines ServiceAccountingCn.ServiceUsage {
10       merge pointcut {
11          merge callee {
12             override component: NewspaperService;
13          }
14       }
15    }
16 }
```

Listing 3.14: The `AccountedNewspaperApp` refining the `BasicNewspaperApp` application with a host, instances and a connector

Now we consider the refinement of an application declaration (Listing 3.14). The goal of the refinement is to extend the newspaper application with an accounting service. To do this, it needs to add the hosts, instances and modules that are required to handle accounting in the application.

In the example in Listing 3.14, the `AccountedNewspaperApp` application refines the `BasicNewspaperApp` from Listing 3.13. The `AccountedNewspaperApp` application inherits all of its members, and adds a host declaration (line 2), a component and a connector instantiation (line 4–5), and an in-line connector declaration (line 8). The `NewspaperAccountingCn` connector that is defined here, is the connector that we considered separately in the section on connector refinement (see Listing 3.12).

Both the `BasicNewspaperApp` and the `AccountedNewspaperApp` have been defined as abstract applications. The reason in both cases is that the hosts in these applications are abstract (see Section 2.2.7.4). To turn an abstract application into a concrete one, we need to refine the application with concrete host definitions.

3.5.4.1 A Concrete Application

```
1  application NewspaperAppDeployment refines AccountedNewspaperApp {
2     host PublishingServer         is "news.cnn.com";
3     host UserManagementServer     is "udb.cnn.com";
4     host ContentManagementServer  is "content.cnn.com";
5     host NewsDeskServer           is "newsdesk.intranet.cnn.com";
6     host Intranet                 is "*.intranet.cnn.com";
7     host StagingServer            is "staging.intranet.cnn.com";

9     NewspaperService       stagingNS  on StagingServer;
10    NewspaperAccountingCn  accConn    on PublishingServer;

12    connector NewspaperAccountingCn refines
          AccountedNewspaperApp.NewspaperAccountingCn {
13       ao-composition ServiceUsage refines
             AccountedNewspaperApp.NewspaperAccountingCn.ServiceUsage {
14          pointcut {
15             callee {
16                host: ! StagingServer;
17             }
18          }
19       }
20    }
21 }
```

Listing 3.15: The `NewsAccountingCn` connector refining its parent with a host condition; in the context of the `NewspaperAppDeployment`.

The non-abstract application specification in Listing 3.15 refines the previous application specification (line 1) with host redefinitions that are no longer abstract. Every inherited abstract host declaration is overridden with a host that defines a physical node (lines 2–7).

In the example in Listing 3.15, a *staging environment* is set up where newspaper employees can perfect layout and editing before publishing to the production server. The `StagingServer` hosts a `NewspaperService` instance (line 9) that serves the newspaper for internal review. As the *accounting* service is not required for the instance on this host, the `NewspaperAccountingCn` connector is refined to exclude the `StagingServer` host (line 16).

3.6 Evaluation

This section presents the evaluation of the ReVew multi-view refinement technique in terms of its support for variability and modularity of architecture description in a case study of the e-Media system. The evaluation consists of three parts:

1. The first part introduces an *extension* of the e-Media system that is used throughout the evaluation (3.6.1). The extension adds various application and deployment variations to the architecture of extended e-Media system.
2. The second part presents the *variability* evaluation (3.6.2). In this evaluation, we compare the description effort required of architects, to realise the variations in the extended case study, between ReVew and two techniques without refinement.
3. The third, and last, part presents the *modularity* evaluation (3.6.3). In this evaluation, we compare how ReVew and the two other techniques deal with the ripple effects of introducing changes into the architecture description.

Finally, we end the section with a conclusion (3.6.4).

Techniques. The two other generic techniques to which we compare ReVew are *Import* and *Flatten*. We describe these technique in a generic way, in terms of the categories that represent the actual techniques that are or could be used in ADLs in the related work.

(a) **Import**. A technique that supports the import of unchanged specifications between two description artefacts. Just like with refinement, all elements within the parent are imported, unless they are overridden. However, *Import* does not support element refinement. Each element (interface, component, connector, AO-Composition) that requires refinement must be manually copied and locally adjusted.

(b) **Flatten**. A technique of manual copy and local adjustment. Architects put all their descriptions, belonging to a single view in the process (e.g. Module), into a single description artefact. Instead of refinement, architects in later views manually copy the specification before adding the necessary changes.

Import is a commonly used technique for reuse between description artefacts in related fields, such as between modules in programming, or scripts in deployment. While an application description can be considered a loose aggregation of various architectural elements, a component, connector and AO-composition have a more rigid structure, that is often hard to reconcile with *import*. Therefore, we believe our distinction between, on the one hand, the application artefact and, on the other, the component, connector and AO-composition artefacts to be fair. Furthermore, we know of no ADL that applies *import* at a granularity level other than the application or system description.

Measuring. In both the variability and the modularity evaluations, we use the Lines of Code metric (LOC), to estimate architect effort, in an absolute as well as a relative comparison.

Unfortunately, we cannot compare the ReVew, *Import*, and *Flatten* techniques directly. To be able to measure lines of code, the techniques will need to be used in conjunction with an ADL. For ReVew, we have selected MViewADL as the ADL to which we apply the technique. Consequently, we apply *Import* and *Flatten* to MViewADL as well, to avoid representational differences from skewing the measurements.

We only count the lines of code of the resulting description artefact of an element, after the changes or variations have been implemented. Furthermore, we count the entire artefact to estimate the effort because (1) the architect needs to consider the entire artefact to decide where to implement the change, and (2) change taints an artefact in its entirety, as architects are not limited to any specific part of the description.

A *description artefact* is the physical counterpart (e.g. a file) of an architectural element (e.g. a component, a connector, an application). We will often shorten *description artefact* to descriptor, or artefact in this evaluation section.

3.6.1 Architectural Variations in the e-Media Case Study

We used MViewADL to specify the architectural structure of the e-Media system. The e-Media system consists of a basic Newspaper service that is accompanied by four additional services, namely *Accounting*, *Personalization*, *User tracking*, and *Authentication*, by means of aspect-oriented techniques.

A well designed complex distributed system supports variations to the desired feature set, as well as variations based on the deployment into different enterprise environments. Supporting variation in the feature set implies design-level alternatives where architects are able to easily combine desired configurations. This enables architects to adapt the architecture to changes in the requirements over time. Supporting variation in the environment implies making sure that architects can easily adapt the architecture to changes in the deployment environment or to deploy it in a entirely different environment.

The e-Media system is extended with the following four variations in the two dimensions of feature set and deployment environment. We give a brief overview of these four variations, before describing them in detail in the remainder of this subsection.

1. Variations in the feature set
    (a) **Load balancing.** Architects extend the e-Media system with an application-level load balancing solution. Selecting this variation implies creating an e-Media system with an additional Newspaper service during the design of the architecture.
    (b) **Non-personalized.** Architects remove the Personalization and User tracking services from the e-Media system. Selecting this variation implies creating an e-Media system that consists of the basic Newspaper service, only extended with Accounting and Authentication.
2. Variations in the deployment environment
    (a) **Multiple environments.** Architects design the e-Media system to support deployment in two environments which we will call CNN and NYTimes. This variation implies describing a specific deployment configuration for each of the environments that must be tuned to its specific networking layout.
    (b) **Staging environment.** Architects design a deployment-level extension of the e-Media system to support a staging environment.

In summary, variation in the feature set results in the need to express design-level alternatives where architects are able to easily combine desired configurations. ADLs must support the adaptation of the architecture to changes from the requirements. While the right granularity in interface- and component design comes a long way, we believe the lack of support for variability in higher-level architecture

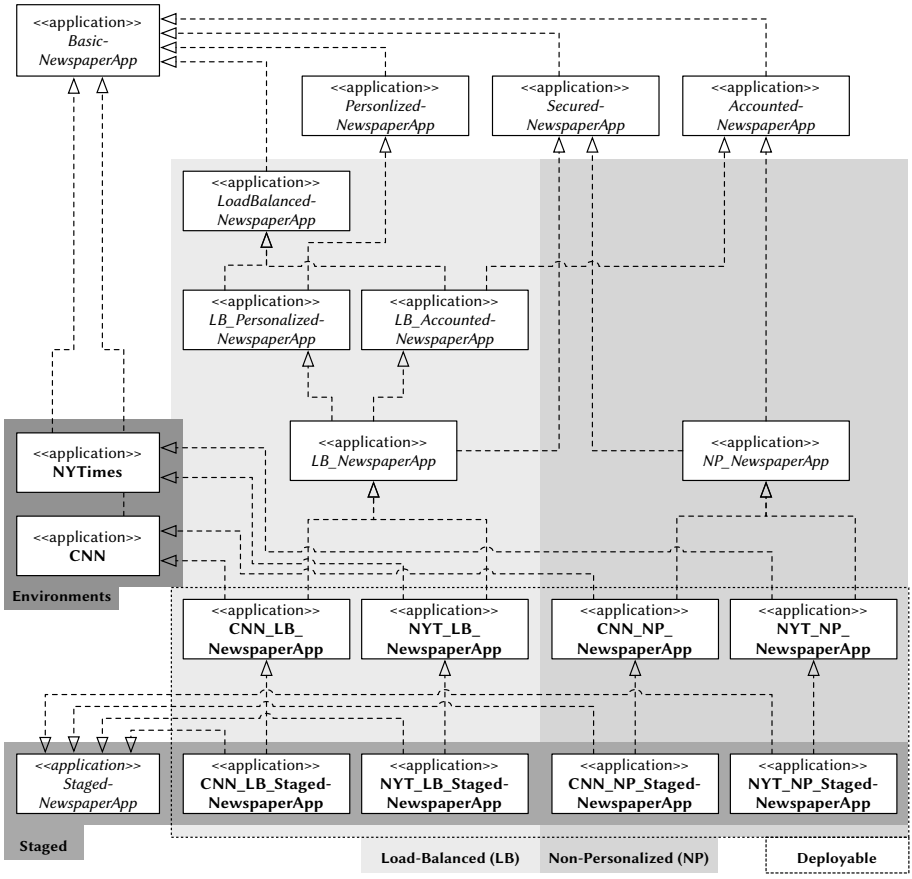


Figure 3.3: ReVew refinement supports the structured description of the many variations of the e-Media system

```
1 abstract application LoadBalancedNewspaperApp refines BasicNewspaperApp {
2   host Proxy;
3   LoadBalancedNS    lbNS      on Proxy;
4   LoadBalancedNSCn  lbNSConn  on Proxy;

6   host PublishingServerAlt2;
7   NewspaperService  nsAlt2    on PublishingServerAlt2;
8 }
```

Listing 3.16: The `LoadBalancedNewspaperApp` application description, refining the `BasicNewspaperApp` application with hosts and instances.

descriptions still hinders architecture evolution. Furthermore, ADLs must also support variation in the environment. This implies making sure that architects can easily adapt the architecture to changes in the deployment environment or to deploy it in a entirely different environment.

Figure 3.3 illustrates how these variations are structured in the architecture description using ReVew refinement. The combination of the proposed variations (load balanced and non-personalized, CNN and NYTimes, as well as a staged and non-staged version) results in eight distinct variants of the application.

A figure that illustrates the variations under the *Import* technique, would look exactly like Figure 3.3, but with import relations, instead of refinement relations. However, this apparent similarity is limited to the granularity level of the application. Once inside these application artefacts, import is of no use, because it cannot handle iterative changes to component, connectors or AO-Compositions.

Figure 3.4 shows how the *Flatten* technique requires the creation of eight variants of the description to capture all of the desired combinations. What the rainbow of grey boxes tries to convey is that closely related variants, like `CNN_LB_Newspaper` and `NYT_LB_Newspaper`, almost entirely overlap (same greys), except for some very specific points (different greys). On the other hand, the difference between variants that have not much in common, like `CNN_LB_Newspaper` and `NYT_NP_StagedNewspaper`, are more outspoken.

3.6.1.1 Load Balancing (variation 1a)

The Load Balancing variation deals with the quality requirement of availability of the Newspaper service. The application-level load balancer in this case study is motivated by the work of Othman, et al. [OBS03] on a Load Balancing service in

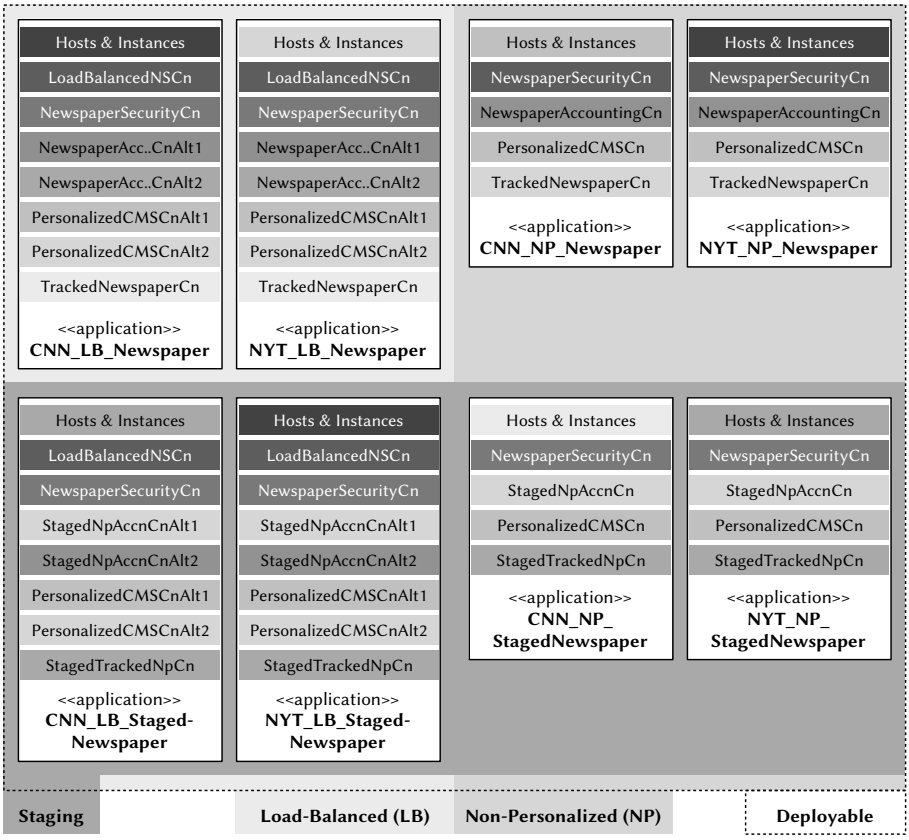


Figure 3.4: The *Flatten* technique results in eight strongly overlapping variants in which the variations are hard-coded

Adaptive Middleware. Application-level load balancing provides the flexibility of making content-aware decisions based on application-defined metrics, at runtime.

Figure 3.3 shows how the e-Media architecture descriptions are extended to support the load balancing requirement. The `LoadBalancedNewspaperApp` application description (detailed in Listing 3.16) instantiates a `LoadBalancerNS` component and a `LoadBalancingNSCn` AO-connector on the `Proxy` host. It declares an alternative `PublishingServer` host and puts an additional `NewspaperService` instance on it.

```
1  abstract application LB_AccountedNewspaperApp refines AccountedNewspaperApp,
2                                  LoadBalancedNewspaperApp {
3    host AccountingServerAlt2;
4    AccountingService          accsAlt2      on AccountingServerAlt2;
5    NewspaperAccountingCnAlt1  accnConn      on PublishingServer;
6    NewspaperAccountingCnAlt2  accnConnAlt2  on PublishingServerAlt2;
7
8    connector NewspaperAccountingCnAlt1 refines NewspaperAccountingCn {
9      ao-composition ServiceUsage refines NewspaperAccountingCn.ServiceUsage {
10       pointcut { callee { override instance: ns; } }
11       advice { instance: accs; }
12     }
13   }
14   connector NewspaperAccountingCnAlt2 refines NewspaperAccountingCn {
15     ao-composition ServiceUsage refines NewspaperAccountingCn.ServiceUsage {
16       pointcut { callee { override instance: nsAlt2; } }
17       advice { instance: accsAlt2; }
18     }
19   }
20 }
```

Listing 3.17: The `LB_AccountedNewspaperApp` application description, refining `AccountedNewspaperApp` and `LoadBalancedNewspaperApp` with hosts, instances and connector refinements for load balancing.

The load balancing connector uses the aspect-oriented interception mechanism to avoid making invasive changes to existing components [Lag09]. The connector ensures the load balancer intercepts all calls of clients to the `NewspaperService` component. Based on their load and availability, it delegates these calls to one of two `NewspaperService` instances.

The Accounting and Personalization services now serve both NewspaperService instances. To avoid that these services become the bottleneck, `AccountingService` and `PersonalizationService` need to be load balanced too. This is done separately for each service in the `LB_AccountedNewspaperApp` and in the `LB_Personalized-NewspaperApp` application descriptions. As both descriptions are similar, we explain only the one of Accounting.

Listing 3.17 shows `LB_AccountedNewspaperApp` refining the `AccountedNewspaper-App` and `LoadBalancedNewspaperApp` applications. It defines an additional host and puts a new `AccountingService` instance on it. Because the Accounting service is triggered by means of an AO-composition, that composition needs to be refined to have each Accounting instance (lines 11 and 17) intercept calls to the corresponding `NewspaperService` instance (lines 10 and 16). All of these changes can be modularized within a single description by means of ReVew refinement.

```
1 abstract application LB_NewspaperApp refines LB_AccountedNewspaperApp,
      LB_PersonalizedNewspaperApp, SecuredNewspaperApp {}

3 abstract application NP_NewspaperApp refines AccountedNewspaperApp,
      SecuredNewspaperApp {}
```

Listing 3.18: The `LB_NewspaperApp` and `NP_NewspaperApp` declarations, combining three and two other apps through refinement, respectively.numbers

Finally, Listing 3.18 shows how `LB_NewspaperApp` combines the application descriptions `LB_AccountedNewspaperApp`, `LB_PersonalizedNewspaperApp` and `SecuredNewspaperApp` using refinement.

3.6.1.2 Excluding Personalization (variation 1b)

While the load balancing variation is about including an additional feature, the personalization variation deals with excluding the Personalization service from the Newspaper application. Personalization is an expensive service in terms of system load. Deploying a variation of the application without this feature may be desirable in particular cases. Furthermore, the Personalization service is also unsuited if a publisher wants to roll out a statically designed site layout.

Creating an application description that does not include Personalization is straightforward. Instead of refining all three parent applications, the `PersonalizedNewspaperApp` parent is not included in the declaration. The resulting application description, called `NP_NewspaperApp`, is shown in Listing 3.18.

3.6.1.3 Changes in the Environment (variation 2a)

Deploying distributed systems of a certain size and complexity (like the e-Media system) is a non-trivial task. The choice for a certain deployment configuration has an impact on the quality and execution of the system. Consequently, this configuration must be easy to adapt as the deployment environment is often subject to change. These changes range from replacing a single host to deploying in an entirely different environment at another organization. The e-Media case study defines two different deployment environments, `CNN` and `NYTimes`. Listing 3.19 shows the deployment configuration of the `BasicNewspaperApp` in both environments.

```
application NYTimes refines BasicNewspaperApp {
   host PublishingServer         is "published.nytimes.com";
   host UserManagementServer     is "customers.nytimes.com";
   host ContentManagementServer  is "data.nytimes.com";
   host NewsDeskServer           is "workers.nytimes.com";
   host Intranet                 is "*.private.nytimes.com";
}

application CNN refines BasicNewspaperApp {
   host PublishingServer         is "news.cnn.com";
  // etc.
}
```

Listing 3.19: Application deployment descriptions for the `BasicNewspaperApp` in the `CNN` and `NYTimes` environments

```
application NYT_NP_NewspaperApp refines NP_NewspaperApp, NYTimes {
  host AccountingServer  is "accounting.nytimes.com";
  host AuthServer        is "auth.nytimes.com";

  NewspaperSecurityCn    secnConn  on PublishingServer;

  connector NewspaperSecurityCn refines
       SecuredNewspaperApp.NewspaperSecurityCn {
    ao-composition ServiceUsage refines
         SecuredNewspaperApp.NewspaperSecurityCn.ServiceUsage {
      pointcut { caller { merge host: ! Intranet; } }
    }
  }
}
```

Listing 3.20: The `NYT_NP_NewspaperApp` description, defining additional hosts and refining the Authentication connector.

As shown in Figure 3.3, the `CNN` and `NYTimes` descriptions are reused, by means of refinement, in a total of eight application variants. The `NYT_NP_NewspaperApp` variant is shown in Listing 3.20. It represents the non-personalized Newspaper application that is deployable at the NYTimes environment. It refines the `NP_NewspaperApp` description from Listing 3.18 with additional host definitions and a refinement of the `NewspaperSecurityCn` connector. The hosts definitions are not part of the `NYTimes` description because they are declared as a part of the Accounting and Authentication services and do not belong to the basic Newspaper application. The connector refinement is required to limit Authentication with the Newspaper service to hosts outside of the intranet.

```
1  abstract application StagedNewspaperApp {
2    host StagingServer;
3    NewspaperService  stagingNS  on StagingServer;

5    abstract connector StagedNewspaperStemCn {
6      abstract ao-composition StagedStemComposition {
7        pointcut { callee { merge host: ! StagingServer; } }
8      }
9    }
10 }
```

Listing 3.21: The `StagedNewspaperApp` description, defining an additional host, a `NewspaperService` instance and the stem staging connector.

```
1  application NYT_NP_StagedNewspaperApp refines StagedNewspaperApp,
       NYT_NP_NewspaperApp {
2    host StagingServer is "staging.intranet.nytimes.com";
3    StagedNpAccnCn  accnConn  on PublishingServer;

5    connector StagedNpAccnCn refines
6        NewspaperAccountingCn, StagedNewspaperStemCn {
7      ao-composition ServiceUsage refines
8          NewspaperAccountingCn.ServiceUsage, StagedStemComposition {}
9    }
10 }
```

Listing 3.22: The `NYT_NP_StagedNewspaperApp` application, applying staging to the non-personalized NYTimes Newspaper application.

3.6.1.4 Staging Environment (variation 2b)

The staging environment variation involves extending the deployment configuration with an additional `NewspaperService` that is used as an internal test setup for verifying content and website layout. This is similar to the staging solution we explained in Listing 3.15, but it is applied to the `CNN` and `NYTimes` environments. Because staging is now applied to four application descriptions instead of one, the common `NewspaperService` instantiation and connector definition (called `StagedNewspaperStemCn`) are now separated out into `StagedNewspaperApp`.

To apply staging to an application, we employ *multiple refinement* of the connector and the AO-composition as shown in Listing 3.22. This comes down to merging both pointcuts, where the *"exclude StagingServer"* host-condition (Listing 3.21, line 7) is added to the pointcut that is defined in `NewspaperAccountingCn`.

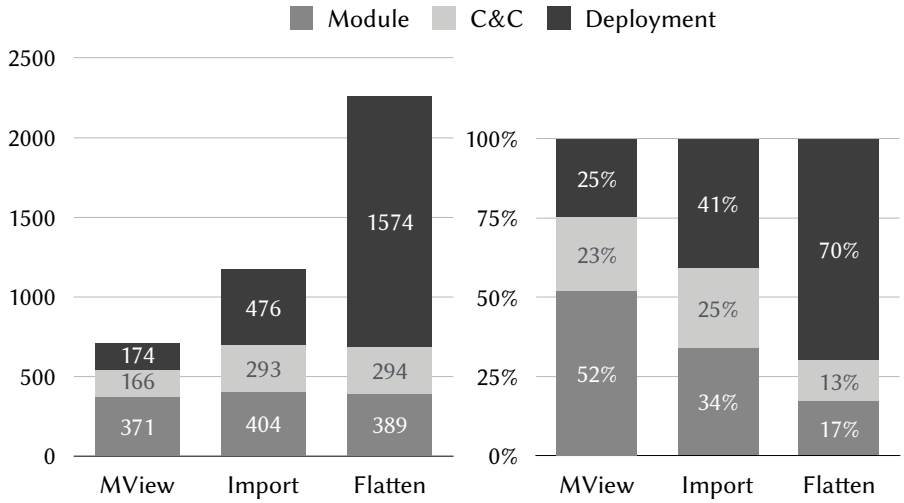


Figure 3.5: ReVew demands less architect effort to realise all eight architecture variations *(left)* and, more specifically, demands a lot less effort of the deployment architect (*right*)

3.6.2 Evaluating The Impact of Variability on the Description Effort of Architects

The results of this evaluation are presented in Figure 3.5. To realize all eight variants, the total lines of description per technique, in the order as presented on the left graph, are 711, 1173, and 2257 LOC. This amounts to ReVew having, respectively, 39% and 69% fewer LOC than *Import* and *Flatten*. The substantial difference in size can be attributed to the potential for reuse in ReVew vs. *Import* and *Flatten*, with this last technique demonstrating no reuse at all.

The Module view descriptions consist of a large common part of 264 LOC — shared among techniques— that defines the interfaces and components. The remaining 107 (15%), 140 (11%), and 125 (5%) LOC, for ReVew, *Import*, and *Flatten* respectively, are concerned with the description of the basic Module-level connectors. This is a relatively big part for ReVew, because it is in these connectors that the reusable bits are defined (e.g. Listing 3.10, 3.11). This allows later descriptions to be defined as a delta. However, because of internal reuse, ReVew can still express these in fewer LOC in comparison to the other techniques.

A similar conclusion applies to the assembly architects in the Component-and-connector view: 166 LOC for ReVew vs 293 LOC and 294 LOC for the other techniques. The relative effort is respectively 23%, vs 25% and 13%, for *Import* and *Flatten* respectively. Because the assemblers that use either ReVew or *Import* build modular descriptions for each of the additional Newspaper services, many application- and connector descriptions are produced. This explains the bad result of *Import*, as it cannot reuse a previous connector during refinement. Also, because *Flatten* cannot benefit from the variability of such a structure, assemblers applying this technique need to manually specify the two application variants —a smaller effort that yields a less flexible design.

It is the job of a deployer to refine the descriptions with host allocations and deployment-level connector refinements. In the case of the e-Media system, deploying the two application variants (load balanced and non-personalized) in two environments (`CNN` and `NYTimes`) in a staged and non-staged setup. The involved effort is considerably less with ReVew: 174 vs. 476 and 1574 LOC. The relative effort for ReVew is 25% vs. 41% and 70% for *Import* and *Flatten* respectively. *Flatten* requires the most effort, as the deployers need to copy the entire component-and-connector view application description eight times, before carefully adding changes. *Import* does a lot better, as the adjustments of the deployers are limited to the host allocations and the changes to entire copies of only a limited number of connector descriptions. ReVew performs best, as refinement enables the changes to connectors to be done in terms of the smaller delta. Both ReVew and *Import* require a total of 13 connectors, while *Flatten* needs 44 similar, but separate, connectors.

3.6.3 Evaluating Modularity in the Change Scenarios

In this part of the evaluation we apply a number of change scenarios to the existing extended e-Media system architecture. A change scenario describes a modification to the architecture —the result of a change in the requirements— and describes what the consequences are of that modification across the architecture (ripple effects). We have selected three distinct scenarios that belong to different views. These scenarios follow from a focus shift in the publishing industry towards a broader kind of content delivery that can even include third party services.

1. **A Module-view change** where the `NewsBrowse` interface is generalized to reflect a change from fetching articles to fetching arbitrary content items.

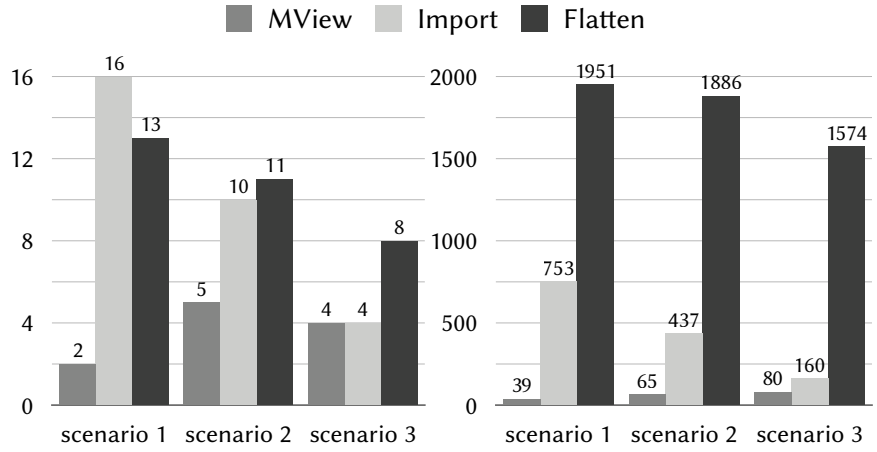


Figure 3.6: Containing change ripple effects with modularity: change affects fewer (or equal) descriptors when using ReVew *(left)* and fewer lines of descriptor code (*right*), across all three scenarios

2. **A Component-and-connector-view change** where the `NewspaperService` component is replaced with a more general `ContentService` component.

3. **A Deployment-view change**, for the CNN deployment, where the `AccountingServer` host is allocated to the same physical host as the `PublishingServer`, instead of a dedicated one.

Figure 3.6 shows the resulting ripple effects caused by the three scenarios in terms of the number of affected descriptors and the amount of description code (in LOC) in these descriptors. The modularity of ReVew is able to contain the impact of these modifications better, which is clearly visible in both graphs.

In scenario 1, ReVew is able to contain the interface modification within 2 descriptors (5%) and 39 lines of affected description (5%). Because this change affects an essential pointcut description in the design, *Import* and *Flatten* fare much worse. *Import* amounts to 16 descriptors and 753 LOC, while *Flatten* amounts to 13 descriptors and 1951 LOC. The reason is that they are unable to reuse previous connector definitions. The description code that is located in the affected descriptors amounts to 64% and 86% of the total code, for *Import* and *Flatten*, respectively.

In scenario 2, the replacement of an important component affects 5 descriptors (12%) and 65 lines of description (9%). This is in contrast with *Import* where 10 (25%) descriptors and 437 (37%) LOC are affected by the change. *Flatten* fares also worse with 11 (39%) descriptors and 1886 LOC (83%) affected by the change. The impact of this change is large for the *Import* case, because this component is used to constrain a pointcut in parts of the design. In addition, it is the duplicated instantiations and allocations of the component in several application variants and deployments that further worsen *Flatten*'s results.

The changes in scenarios one and two can be considered quite invasive. They involve, respectively, modifying an interface and substituting a component throughout the entire architecture. The third scenario, modifying the deployment, is more in line with how the description of an architecture is generally expected to change in the short term. This non-invasiveness is noticeable from the number of descriptors that are affected for scenario 3, at least for *Import* and *Flatten*. ReVew and *Import* contain the modification in 4 descriptors (10%) and 80 (11%) and 160 LOC (13%), respectively. *Flatten*, again, scores worst of the three with 8 descriptors (29%) and 1574 LOC (70%). This result would be better for ReVew and *Import*, if the host allocations such as that of `AccountingServer` in Listing 3.20, would have been defined in separate descriptors per deployment variation.

3.6.4 Conclusion of the Evaluation

We have compared ReVew, when applied to MViewADL, to two other techniques, *Import* and *Flatten*, in terms of the architect effort (measured in LOC) in a variability and modularity evaluation.

In the variability part, ReVew is shown demanding less architect effort to realise all eight architecture variations. In addition, ReVew demands a lot less effort of the allocation architect when realising the four deployment environment variations.

In the modularity part, ReVew is able to better contain the ripple effects in two change scenarios, resulting in far fewer affected descriptor artefacts. In terms of the lines of description inside these artefacts, ReVew affects far fewer lines, in all three scenarios.

The results of both parts of this evaluation are from an architecture that describes just eight variations. Consider what this means when this number reaches twenty or thirty. Modular ADLs are necessary to manage this kind of variability and evolution of software architectures.

Appendix B.1, B.2, B.3, and B.4 present a detailed overview of the raw evaluation data. More information on the evaluation is available online [OvDLJ14].

3.7 Conclusion

The software architecture of a distributed system is created by means of a process that is driven by multiple architects with varying expertise. The state-of-the-art in architecture description languages offers little support for identifying the architect and the requirement that lead to a specific contribution. This results in a loss of traceability, modularity and variability within the architecture description of a software system.

Multi-view refinement is a concept for the modularization architecture description. The concept describes the idea of stepwise refinement of the description by enabling architects to contribute separately, at the appropriate time, and in terms of the concepts that match their expertise. ReVew is our technique that implements the concept of multi-view refinement for the use in ADLs.

We validate the ReVew technique by applying it successfully to various elements with MViewADL. We follow this up by an evaluation of ReVewin MViewADL where we verify our modularity and variability claims on an extended e-Media case study.

Finally, we revisit the requirements (Section 3.3) to see how MViewADL fares in the comparison with the related work. Before the integration of ReVew, MViewADL already offered full support for *open definitions* and *multiple models and views*. Integrating the refinement technique does not change this, because it does not add any new elements. Rather, it adds a refinement relation to the elements in the language. This relation allows models of the same element type to refine each other. In addition, refinement supports models the belong to different views (Figure 3.1). This allows us to conclude that MViewADL+ has full support for the *model and view relations* requirement.

Discussion

In order to achieve the vision put forward in Figure 3.1, MViewADL+ reuses concepts and techniques that are also present in the related work (see Section 3.3): multi-model and multi-view descriptions, modularization and adaptation of com-

plex compositions [Lag09], and model relations, in particular, the refinement relation.

The challenge with MViewADL was integrating these into a single consistent language model that would achieve our requirements. Take for example *refinement*, which we use throughout the model because it fits naturally with the process of architectural refinement. Integrating it meant figuring out the meaning of refinement for each concept to which it was applied. We quickly learned that *override* alone was not enough when refining, leading to the introduction of the *merge* operation.

On the other hand, there is still room for the MViewADL and ReVew to further evolve. Currently, refinement has complete access to a parent's description. This may not always be desirable. One way to address this is through accessibility modifiers. Also, while refinement is broadly applicable, the syntax and semantics of the ADL will need to be extended to support additional views.

Another issue is related to behavioural subtyping and what happens to child elements when a parent element is changed. First, architecture description is not programming code, and ReVew is a more general technique than inheritance. In MViewADL alone it has been applied to four different kinds of declarations. Second, it is yet unclear if the problem can be solved in the language at all. It is possible that the proper tool-support could alleviate the problem uncontrolled changes to existing elements in the description.

Finally, we are able to support and express refinement between models of different views in MViewADL, because these views have been designed with a common goal: the description of distributed systems. Keeping the language and its models consistent when adding arbitrary, independent views might prove challenging as well.

CHAPTER 4

Platform and Tool Support

"An elegant architectural model is of limited value unless it can be converted into a running application."

— Medvidović and Taylor [MT00]

MViewADL is supported by tools that assist in the creation and validation of MViewADL descriptions. In addition, to further increase the usefulness of the ADL, these tools support the production of implementation code for various middleware platforms, based on architecture descriptions in MViewADL. We start with a discussion of the two goals of this chapter.

4.1 Goals

The goals of this chapter are two-fold. The first goal is about assisting in the production of code for multiple middleware platforms from MViewADL descriptions. While the second goal is about supporting MViewADL with tools to get the most out of using the language. These goals are not mutually exclusive, as the tool support will make use of the knowledge on code production for the purpose of automatic code generation.

Goal 1. Assisting in the production of code

As Medvidović and Taylor state in their quote under the heading of this chapter, the purpose of a software architecture ultimately is to describe what is to be built, or alternatively, to assess that the system that is required, cannot be built. Architecture description specifies elements such as interfaces, components, connectors, configurations, hosts, etc. Most of these elements are not exclusive to software architecture, as we discussed in Chapter 2 on MViewADL. This means that it is possible to take many of these architectural elements and turn them into the corresponding implementation and deployment artefacts. The completeness of the artefacts that can be produced, based on the knowledge in the architecture, varies widely, ranging from complete artefacts (interfaces) to mere skeletons (component).

A systematic way of producing code artefacts from architecture description is often very useful to developers. It requires architects to be more accurate with their descriptions, and it requires less interpretation by developers, reducing the possibility of errors.

The production of code from architectures is one of several techniques to minimize *architecture erosion* [dSB12]. Architecture erosion is the overall degradation in quality of architecture specifications as a consequence of software evolution, i.e. changing requirements or conditions. An essential tool for controlling architectural erosion is *traceability* across requirements, architecture and implementation. Traceability is useful for tracing changes in the architecture to the corresponding elements in the implementation, and vice-versa, tracing issues in the implementation back to the weaknesses in the architecture.

Erosion is then controlled by carefully evolving the architecture and the resulting software system as a whole. The role of systematic code production, or tool support in general, is that it is an incentive to keep maintaining the architecture.

Goal 2. Tool Support for MViewADL

Architecture description languages are generally created to support architects, and by extension developers, (a) in building better architectures through analysis and validation and (b) in better documenting the architecture that they are building. However, to succeed in doing that, it is not enough for architects to know the language, they must also have the proper tools to support them [MLM+13].

Medvidović and Taylor have categorized tool support [MT00] into active specification, multiple views, analysis, refinement, implementation generation, and dynamism. While it would be a lofty goal to provide tool support for all of these, we have chosen to focus on those categories that fit best with the goals of MViewADL. These are multiple views, refinement and implementation generation.

In short, the *multiple views* category is about supporting different stakeholders — in this work we focus on a subset of stakeholders: developers— in accessing the architecture descriptions from their particular view, while ensuring consistency between the views. *Refinement* is about supporting the refinement of the architecture across levels of detail. Finally, the *implementation generation* category states that the goal of any modelling endeavour in software development is to produce an executable system. Relying on a manual process may result in problems with consistency and traceability between the architecture and its implementation.

Some of these goals might seem familiar, as MViewADL with ReVew already offers support for multiple views and refinement in the language. Yet, these language features are hard to use without the proper tool support.

We address each goal in the following two sections, *Maintaining Traceability in Implementation* (Section 4.2) and *Description and Code Generation Tool Support for MViewADL* (Section 4.3). This is followed by an overview of the related work and a discussion on validating our tool support.

4.2 Maintaining Traceability in Implementation

In this section, we give an overview of the strategies for converting MViewADL descriptions to implementation artefacts, while providing a suitable level of modularity and variability to maintain traceability between architecture and implementation. This overview is based on the lessons learned and subsequent refinements from our work on building a digital publishing platform [VOTV10].

We categorize these strategies into *Components and Interfaces*, *Connectors and Compositions*, and *Platform-specific Strategies*.

4.2.1 Components and Interfaces

The concept of an interface at the levels of software architecture and implementation are essentially the same: "An interface remains a description with a unique name that lists the services, operations or methods that define its purpose" (see Section 2.2.2 on page 31). As such, its representation at the implementation level is often just a straightforward transfer involving some minor syntactical changes. Furthermore, inheritance can be used to retain refinement relations between interfaces in the architecture description.

```
1 component NewspaperService {
2    provide {
3       NewsBrowse
4    }
5    require {
6       ContentBrowsing
7    }
8 }
```

Listing 4.1: The essence of a component description lies in the interfaces it provides and requires

In architecture, a component defines a contract that states those interfaces that it provides and requires. At the level of implementation, the component retains the same notion of a contract. In addition, a component will also encapsulate the implementation logic and optional state that realizes the services it provides.

Listing 4.2 shows the JBoss representation of the `NewspaperService` component. It implements the basic NewspaperService contract that was defined in the architectural description of the component, in Listing 4.1. The component provides the `NewsBrowse` interface by *implementing* the source code representation of the interface (line 3), and it requires the `ContentBrowsing` interface by declaring it as a dependency (line 6). At runtime, the platform will resolve a dependency to a suitable component instance by means of a process called *dependency injection*. The Spring implementation of this component is shown in Listing 4.3. It uses different annotations, but the same results can be achieved.

```
/** Defining the functionality to browse for and read articles. */
@Stateless
public class NewspaperService implements NewsBrowse {

   @EJB  /* a required dependency on ContentBrowsing */
   private ContentBrowsing _cms;

   /** Fetch an article with id $contentId$. ... */
   @ServiceUsage
   public ContentItem fetchArticle(ContentItemId contentId) { ... }

   /** Returns a list of $number$ article headlines. ... */
   public List<ArticleEntry> listHeadLines(int number) { ... }

   // ...
}
```

Listing 4.2: The `NewspaperService` JBoss component implements its contract by providing the `NewsBrowse` and requiring `ContentBrowsing`

```
/** Defining the functionality to browse for and read articles. */
@Component
public class NewspaperService implements NewsBrowse {

   @Autowired  /* a required dependency on ContentBrowsing */
   private ContentBrowsing _cms;

   /** Fetch an article with id $contentId$. ... */
   @ServiceUsage
   public ContentItem fetchArticle(ContentItemId contentId) { ... }

   /** Returns a list of $number$ article headlines. ... */
   public List<ArticleEntry> listHeadLines(int number) { ... }

   // ...
}
```

Listing 4.3: The Spring implementation of `NewspaperService` differs slightly in terms of annotations

Listing 4.5 shows one possible implementation of the contract defined by the `ProfiledNewspaperService` description. The `ProfiledNewspaperService` component description is a refinement of the `NewspaperService` description in Listing 4.4. Here, the refinement relation can be successfully retained by means of an inheritance relation in the implementation (line 3).

```
component ProfiledNewspaperService refines NewspaperService {
   provide {
      ConsumerProfile,
      NSManagement
   }
   require {
      UserProfiling
   }
}
```

Listing 4.4: A component that is refined with additional provide and require dependencies

```
 /** An extension to manage the media consumer profile. */
@Stateless
public class ProfiledNewspaperService extends NewspaperService implements
     ConsumerProfile, NSManagement {

   @EJB  /* a required dependency on UserProfiling */
   private UserProfiling _ums;

   /** Subscribe and unsubscribe to the newspaper. */
   public void setSubscription(Period period) { ... }

   // ...
}
```

Listing 4.5: Extending the component's contract with a provision of `ConsumerProfile` and `NSManagement`, and a dependency on `UserProfiling`

We have shown that inheritance in object orientation can be used to retain the modularity provided by ReVew refinement. However, this only works well when ReVew is used to extend the parent element in the child. While something like replacing the provided interfaces of a component is possible in ReVew, it is not supported by inheritance under behavioural subtyping.

4.2.2 Connectors and Compositions

The connector is a concept that comes in many forms and flavours, both at the level of architecture, as well as implementation. Providing a definitive overview of architecture-level connectors and how they can be captured during implementation, is out of the scope of this work. However, we will describe a number of examples from our case study.

4.2.2.1 Implicit Composition

The implicit provide-require connector between two components, where one provides interface *I* and the other requires interface *I*, is implicit in the implementation as well. Using Listing 4.2, we have previously shown that *implementing* the `NewsBrowse` interface conforms to providing that interface. Now, consider a `Client` component that depends on the `NewsBrowse` interface. The following statement in the JBoss implementation of `Client`, conforms to requiring that interface:

```
@EJB
private NewsBrowse _ns;
```

At runtime, this results in the component that requires `NewsBrowse` being provided with an instance of a component that implements this interface. Because there is only one of these components in this case (no ambiguity), it will receive a `NewspaperService` component instance.

4.2.2.2 Ambiguous Composition

Just like during architecture description, a composition ambiguity at the implementation level, i.e. a dependency on an interface that is provided by multiple components or multiple component instances, must be explicitly solved. Platforms typically provide a way to bind dependencies on an interface to specific implementations of that interface.

In JBoss, selecting a specific implementation for an interface is done by means of the *beanName* dependency injection attribute:

```
@EJB(beanName="NewspaperService")
private NewsBrowse _ns;
```

However, if we are dealing with multiple instances of the same component, the *mappedName* attribute should be used instead. Its value refers to the unique name by which each component instance is known to the system —i.e. the name of the instance, not unlike our architectural solution to this problem:

```
@EJB(mappedName="nsMob")
private NewsBrowse _ns;
```

In Spring, we make use of the @Resource annotation to inject a bean based on its JNDI name:

```
@Resource(”nsMob”)
private NewsBrowse _ns;
```

The solutions we provide here all depend on attributes to annotations inside the source code of the application. This has the disadvantage that the code must be recompiled after every modification of an annotation. Fortunately, these platforms also support configuration files that are separate from the code and do not require recompilation. Furthermore, these configurations can be adjusted by application assemblers and deployers at a later stage, leaving the components themselves unchanged. We chose not to use these configuration artefacts to illustrate our solution, because they are large and complex, and, ideally, shouldn't be managed by hand.

4.2.2.3 AO-Composition

As both platforms support aspect-oriented programming (AOP) techniques, MView-ADL AO-Compositions can be expressed at the level of implementation. First we briefly describe AOP in JBoss and Spring, then we discuss how an AO-Composition can be implemented.

AOP in a nutshell. The core concept in AOP is the aspect class, which supports an arbitrary number of pointcut and advice definitions. Definition of these constructs is done by means of annotation bindings or, alternatively, by providing bindings in separate configuration artefacts.

We follow the growing trend in application development to include more platform configuration as simple annotations in the code, unless it hinders maintainability and evolution (see Section 2.2.5.3 on composition ambiguity). The annotation approach leads to source artefacts with the following characteristics:

1. **Aspect:** An aspect is implemented as a plain Java class that is annotated with the `@Aspect` annotation. It can contain multiple pointcut and advice declarations.
2. **Pointcut:** A pointcut is a predicate that matches join points.

- In JBoss, it is declared by means of the `@PointcutDef` annotation, attached to a public, static `Pointcut` field. The predicate is defined as a String.
- In Spring, a pointcut is declared by means of a regular method signature comprising a name and any parameters, that is annotated with the `@Pointcut` declaration which contains a string with the pointcut predicate.

While JBoss supports both the *execution* and *call* pointcut kinds, Spring lacks support for the latter kind. We solve this by using dependency injection to inject a callee-proxy component, instead of relying on AO to intercept the original call (see Section 4.2.3.2).

3. **Advice:** Advice is associated with a pointcut and is run before, after or around the execution (or call) that triggers the pointcut. An advice method is mostly used to delegate to some business behaviour that is implemented by a business component.

   - In JBoss, advice is associated to a pointcut by annotating the advice method with the `@Bind` annotation. The `pointcut` attribute refers to the pointcut declaration by name, and the `type` attribute of the annotation determines the type (`before`, `after`, `around`) of the advice.
   - In Spring, the annotation determines the type of the advice (`@Before`, `@After`, `@Around`) and its `pointcut` attribute refers to the pointcut declaration by name.

The differences in AOP mechanisms between both platform are manageable. This is not surprising, as they are both inspired by AspectJ. However, because Spring AOP is a proxy-based technique, while JBoss AOP is based on class weaving, it is somewhat less powerful and limited to method interception. Fortunately, since the MViewADL join point model is also limited to method calls on components, the impact of these limitations is small.

Implementing an AO-Composition. To explain an illustrate the implementation of an AO-Composition, we continue the running example of Chapter 3. In this chapter, we defined the `NewspaperAccountingCn` connector across four listings (3.10–3.13) using ReVew refinement.

Using the architectural description of the `NewspaperAccountingCn` connector, we were able to develop a JBoss aspect that implements it fully. Listing 4.6 shows the

```
1  package accounting;
2
3  import accounting.Accounting; // other imports cut
4
5  @Aspect
6  public class NewspaperAccountingCn {
7
8     /* valid and invalid hosts */
9     public static String[] VALID_HOSTS = {};
10    public static String[] INVALID_HOSTS = {"staging....cnn.com"};
11
12    /* the pointcut definition */
13    @PointcutDef(
14       "execution(* * -> @ServiceUsage(..)" +
15       "AND class($instanceof(NewsBrowse)) " +
16       "AND class(NewspaperService)"
17    )
18    public static Pointcut newspaperAccounting;
19
20    /* required for advice */
21    @EJB
22    private Accounting _accounting;
23
24    /* advice method */
25    @Bind(
26       pointcut="newspaperAccounting",
27       type=AdviceType.AFTER,
28       cflow="NpAccountingHostConditions")
29    public void chargeForService() {
30       _accounting.chargeForService();
31    }
32 }
```

Listing 4.6: A JBoss connector implementation

resulting JBoss aspect class. The aspect is called `NewspaperAccountingCn`, in line with the name in its architecture description.

The aspect consists of a pointcut and an advice. The pointcut (line 18) is configured by means of the `PointcutDef` annotation. It is described in the JBoss pointcut language, but we can clearly recognize the *execution* of any method that is annotated with the `@ServiceUsage` annotation on a component implementing the `NewsBrowse` interface. This is an implementation of the design in Listing 3.10.

In addition, the pointcut is further constrained with a definition that the component on which the method is executed should be the `NewspaperService` component. This implements the design in Listing 3.12.

```
1  abstract class HostConditions implements DynamicCFlow {
2     private List<String> validHosts, invalidHosts;
3
4     // constructor and getters, setters for (in)validHosts
5
6     /** @return true if should execute; otherwise false */
7     public boolean shouldExecute(Invocation invocation) {
8        String hostname (new ServerInfo()).getHostName();
9        return (validHosts().isempty() || validHosts().contains(hostname))
10             && ! invalidHosts().contains(hostname);
11    }
12 }
13
14 @DynamicCFlowDef
15 public class NpAccountingHostConditions extends HostConditions implements
       DynamicCFlow {
16    public NpAccountingHostConditions() {
17       super(NewspaperAccountingCn.VALID_HOSTS,
18          NewspaperAccountingCn.INVALID_HOSTS);
19    }
20 }
```

Listing 4.7: Runtime Host Condition Evaluation in JBoss

The advice (line 29) consists of a call to a business method on a component implementing the `Accounting` interface[1]. The advice is linked to its pointcut using the `@Bind` (line 25) annotation. It references the name of the pointcut and includes the type of advice: `after`. This adheres the design in Listing 3.11.

The host-conditions on the composition with a callee component are shown on lines 9 and 10. A callee component conforms if it is allocated on a host in the valid list, but not in the invalid list. In this case, the executing component should no be allocated on the host `staging.internal.cnn.com`. This follows the design in Listing 3.13. How to validate whether the executing component is allocated on a host that is consistent with the valid and invalid hosts lists is handled in the following paragraph, named *Verifying Host Conditions via Dynamic CFlow*.

While the solution described here does not attempt to retain the refinement relations between the individual connector refinements in the design, it is possible to use aspect inheritance to separate pointcut definitions, advice definitions and host list declarations from each other in different classes.

[1]While dependency injection in JBoss does not extend into Aspect classes, the necessary JNDI-lookup can be easily implemented with a custom aspect that intercepts access on fields annotated with the EJB annotation in classes annotated with @Aspect.

Verifying Host Conditions via Dynamic CFlow. JBoss and Spring have some limitations that require specific techniques to solve. One of these is the verification of host conditions: JBoss nor Spring, offer support for reasoning about host allocation in the pointcut.

JBoss supports a runtime construct called *dynamic control flow*[2]. A dynamic cflow is attached to an advice method and allows the execution of additional code that may determine whether or not the advice is triggered.

Dynamic CFlow works by means of a class that is annotated with the `@DynamicCFlowDef` annotation. In addition, this class provides an implementation of the `boolean shouldExecute()` method, imposed by the `DynamicCFlow` interface that it implements.

To solve the host validation problem, we propose to developed a generic, abstract dynamic control flow class `HostConditions` that verifies whether or not the executing component is allocated on a host that is consistent with the valid and invalid host lists: the host should be contained in the `validHosts` list, if it is not empty, and should not be contained in the `invalidHosts` list.

In addition, we define a concrete class `NpAccountingHostConditions` that extends the `HostConditions` class. This class initializes the validHosts and invalidHosts lists based on the String arrays in the `NewspaperAccountingCn` aspect (Listing 4.6. It is coupled to the advice by means of the cflow-attribute of the `Bind` annotation (Listing 4.6, line 25). The advice is then only executed based on the result of the `shouldExecute()` method of the `NpAccountingHostConditions` class.

In Spring, a similar solution can be applied. However, because Spring does not support dynamic cflow classes, host validation should be called manually as the first action in the advice body. The result of `shouldExecute()` determines whether the rest of the advice body is executed.

4.2.3 Platform-specific Strategies

This section discusses two solutions to limitations in the JBoss and Spring platforms. These limitations stem from the limited support for distributed AO in both platforms.

[2]Dynamic CFlow in JBoss is completely unrelated to and inconsistent with AspectJ's cflow concept.

4.2.3.1 Aggregating Context via Cooperating Aspects

MViewADL supports the description of a composition in terms of caller and callee distribution constraints simultaneously, e.g. the caller component instance must not be on host A, while the callee should be on host B. Because of the lacking distribution support in JBoss and Spring, an aspect that is deployed on host B has no way of knowing what host the caller is on, and vice-versa. A single aspect, deployed either on host A or on host B, cannot fully capture the MViewADL connector in this example.

To solve this we need two aspects, one on each host, that cooperate to collect the necessary context information. The caller-side aspect collects the data and passes it to the callee-side aspect by means of the invocation context. The receiving aspect aggregates all data and decides whether to allow the original invocation to proceed. This aspect cooperation is only one possible scenario for implementing MViewADL connectors. The following table presents the four generation scenarios (A–D) that address all common cases. The two dimensions are *the pointcut kind* and *caller- and/or callee-conditions*.

	caller	callee	both
call	B	D	D
execution	C	A	C

The scenarios can be summarized as follows: (A.) strict callee-side interception and invocation context, (B.) strict caller-side interception and invocation context, (C.) caller-side interception with caller- and callee-side context, and (D.) callee-side interception with caller- and callee-side context. We discuss the scenarios in more detail in what follows.

Scenario A. In our running example, we showed the transformation of the `NewspaperAccountingCn` connector from the MViewADL specification into a single aspect class with the same name. NewspaperAccounting is a relative simple case in that it combines the execution pointcut kind with exclusively callee-conditions. Both of these conditions have callee-semantics. The aspect is deployed on the callee-side.

Scenario B. Another case that requires a single aspect class is that where the pointcut is of kind call with exclusively caller-conditions. The biggest difference with scenario A. is that the aspect is deployed on the caller-side.

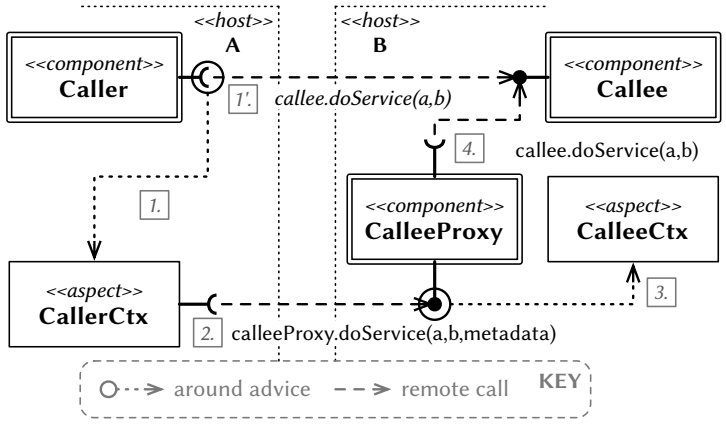


Figure 4.1: Communication between remote aspects through a proxy component

Both scenarios C. and D. require the cooperation of JBoss aspects to verify caller- and callee-conditions.

Scenario C. This scenario has a pointcut of the execution kind and is combined with caller-conditions and optional callee-conditions. This is realized by deploying a caller-side aspect that collects the required context information, and uses the *invocation context* to piggy-back this data to a callee-side aspect. This aspect evaluates both remote and (optionally) local context information, and decides whether to proceed with the original invocation.

Scenario D. This scenario is similar to that of scenario C., in that it requires two aspects cooperating, however, the callee-side aspect will return the control-flow to the caller-aspect, rather than let the invocation proceed. When returning, it will pass on the required callee information. The caller-aspect will in turn aggregate both remote and (optionally) local context information and decide whether to proceed with the invocation.

4.2.3.2 Supporting Distributed AOP

MViewADL supports the description of a composition in terms of caller and callee distribution constraints simultaneously, e.g. the caller component instance must not be on host A, while the callee should be on host B. Because of the lacking distribution support in both JBoss and Spring, an aspect that is deployed on host

B has no way of knowing what host the caller is on, and vice-versa. A single aspect, deployed either on host A or on host B, cannot fully capture the MViewADL connector in this example.

To solve this we need two aspects, one on each host, that cooperate to collect the necessary context information. Both platforms enable aspects to pass along data to each other via the invocation context, but only if these aspects are part of the local interception chain for the same join point. They do not preserve this data when moving from the interception chain at the *call*-side to that of the *execution*-side, in case of a remote invocation.

We can enable the communication between remote aspects by means of the pattern that is shown in Figure 4.1. The original invocation *1'.* of `doService(a,b)` between `Caller` and `Callee` is intercepted by the `CallerCtx` aspect *1.* . The aspect collects the required context data, but instead of proceeding with the invocation, it calls the `CalleeProxy` component `doService(a ,b,metadata)` *2.* . With this call, the aspect pushes the original target object (*callee*), the original arguments (*a,b*) and the additional data (*metadata*). The *execution* of this proxy call is intercepted by the `CalleeCtx` aspect *3.* . This aspect pops the metadata, aggregates it with its own data and decides whether to proceed with the proxy call. If it proceeds, `CalleeProxy` invokes the original call on the target object, with the original arguments *4.*.

4.3 An Overview of MViewADL Tool Support

Tool support for MViewADL serves a double purpose. One is an IDE to assist in the production of MViewADL descriptions. The other purpose is a prototypical code generator that outputs JBoss (and Spring) code. The code generator is essentially a plugin for the IDE.

Tool support for MViewADL consists of four core building blocks, as shown in Figure 4.2. Of these four parts, we built the third, and fourth parts ourselves, on top of the frameworks in one and two:

1. A basic toolset consisting of frameworks such as Eclipse and ANTLR
2. The Chameleon workbench, an extensible language framework and IDE
3. The MViewADL language and IDE as an extension of Chameleon
4. A code generator prototypes for JBoss and Spring

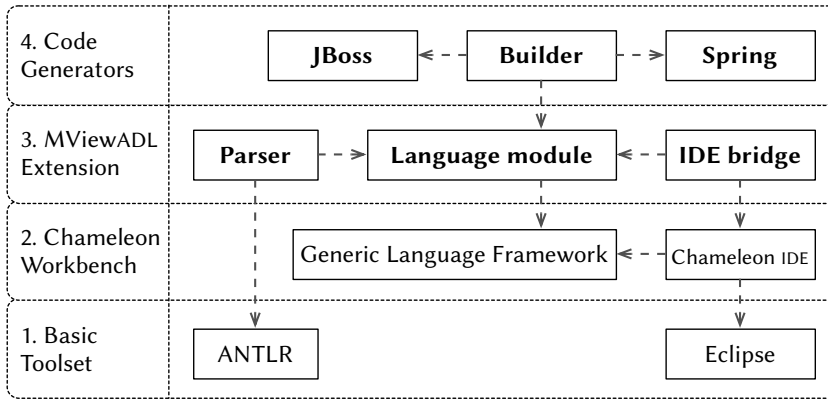


Figure 4.2: The bold IDE building blocks we built ourselves

4.3.1 The Basic Toolset

The base of our tool chain is made up of well-known tools such as Eclipse and ANTLR. Eclipse has been selected for its plugin support, on top of which the Chameleon IDE is constructed.

We use ANTLR [PQ95] to construct a parser for MViewADL descriptions, because it is suitable for parsing our language, it is still being actively developed, and it integrates well with Java (and Eclipse).

4.3.2 The Chameleon Workbench

Chameleon is a language workbench that focuses on eliminating redundancy in the semantics of similar language constructs [vDSJ12]. Language workbenches facilitate the development of languages. The semantics of the language are specified in a domain-specific language that is used to generate tools such as compilers and IDEs.

Instead of using a domain-specific language, classes for language constructs are implemented by hand, and standard object-oriented programming is used to maximize code reuse. By defining a framework of abstract and generic language constructs, Chameleon avoids a lot of code duplication, and enables modular language extensions. These abstractions are further used to provide generic IDE

support. This allows any language developed with the workbench to have IDE support with only a few lines of code.

The difference between Chameleon and similar language workbenches is that these focus on the conciseness of the semantics specifications, but not on eliminating redundancy in similar language constructs. This allowed us to reuse core behaviour such as name lookup, rule verification and IDE support, and focus on designing and implementing the specifics of our language.

4.3.3 The MViewADL Extension

The MViewADL extension is built on top of the Chameleon workbench and consists of a language module, a parser for text-based MViewADL descriptions, and an IDE bridge that sets up the Chameleon IDE for use with MViewADL.

Language Module. The language module is basically an implementation of all the concepts and techniques that we have introduced and discussed in Chapters 2 and 3. We have implemented elements such as the interface, the component, the connector, OO- as well as AO-composition, the application, the host, etc. We offer complete support for the ReVew refinement technique across all applicable language elements.

Parser. The parser for MViewADL descriptions lies at the basis of the entire tool chain. We use ANTLR (Figure 4.3) to generate a parser for our language. We do this by designing a grammar in EBNF (Extended Backus–Naur Form), which ANTLR uses to output a parser program (consisting of a parser and a lexer). Figure 4.3 shows the EBNF specification for the component element of our language. The parser takes in MViewADL descriptions and outputs an AST (abstract syntax tree) in the form of in-memory objects of the language module. This AST is later used by the IDE and various code generators.

IDE Bridge. As far as tool support goes, a dedicated IDE for your language is invaluable. As MViewADL descriptions are textual, our tool support is focussed on features that are relevant to code editing tools. Nevertheless we do want to stress that we value the benefits of graphical representations in basic communication, as apparent from our use of illustrations in Chapter 2. Because

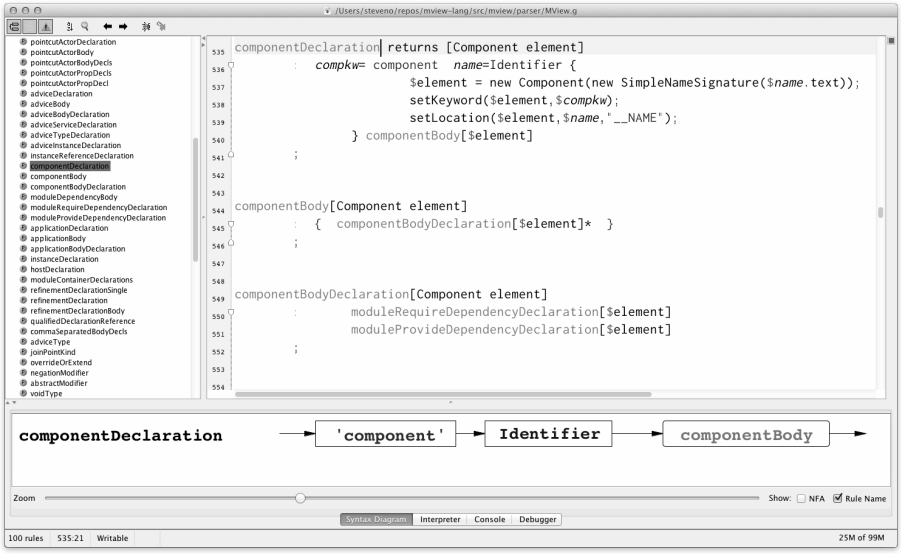


Figure 4.3: The third-party ANTLR tool allows us to design and generate a parser for MViewADL

MViewADL makes use of the typical architectural views of module, component-and-connector, and allocation, and UML supports these views, it can be used to graphically model descriptions in MViewADL. UML 2.0 comes with a set of diagram types —which it calls UML *views*[3]— to model these particular aspects of the design of a system [RJB05].

The main features that the MViewADL IDE supports are:

- Syntax highlighting and code browsing
- Code outline and meta-model browsing
- Validation against language semantics
- Error reporting and tracing

Figure 4.4 shows the MViewADL IDE. The image illustrates features such as *syntax highlighting* and *error reporting*. The error in this example is on the highlighted

[3] A view is simply a subset of UML modelling constructs that represents one aspect of a system. [RJB05]

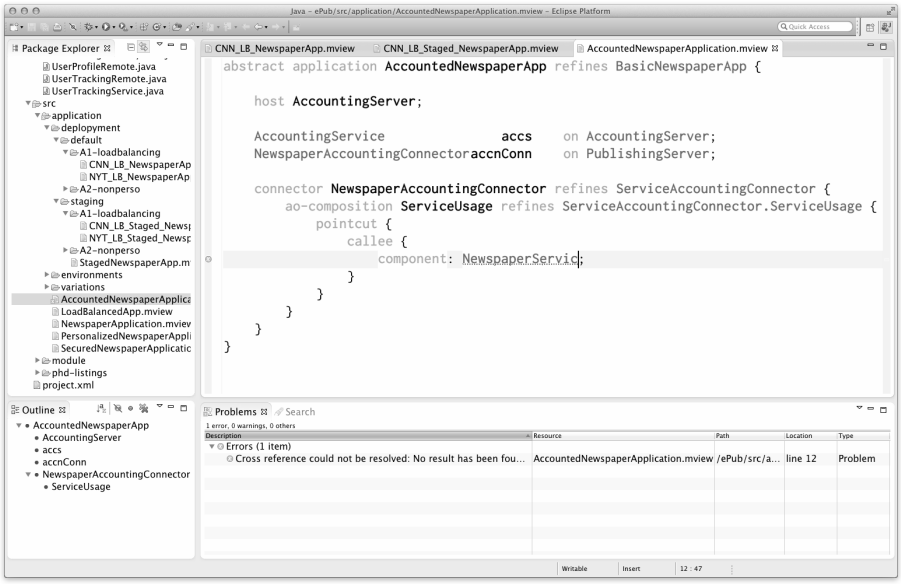


Figure 4.4: Support for the typical code editing features like highlighting, code walking and error reporting

line. The reference to the `NewspaperService` component type contains a typing error, it is missing an "e" at the end. Because of this, the IDE cannot match the string with an existing type. The string is underlined and an error entry is added to the *problems viewpane* at the bottom, where you can trace the error back to its location in the description artefact.

The IDE supports *code browsing*, which means that every reference to a declaration (components, connectors, interfaces, methods, hosts, instances, applications, etc.) can be *ctrl-clicked* to navigate to where that declaration is defined. In the figure, declarations are shown in dark grey, references are light grey, and keywords are the lightest colour grey.

The figure also shows an *outline viewpane* on the bottom-left side. Every entry in this pane refers to the definition of a declaration. It also indicated which declarations are nested within other declarations. The IDE will also display *errors against the syntax and the semantics* of the language, e.g. invalid or misspelled keywords, missing elements or declarations (no advice in an AO-composition, no instances in an application, etc.), …

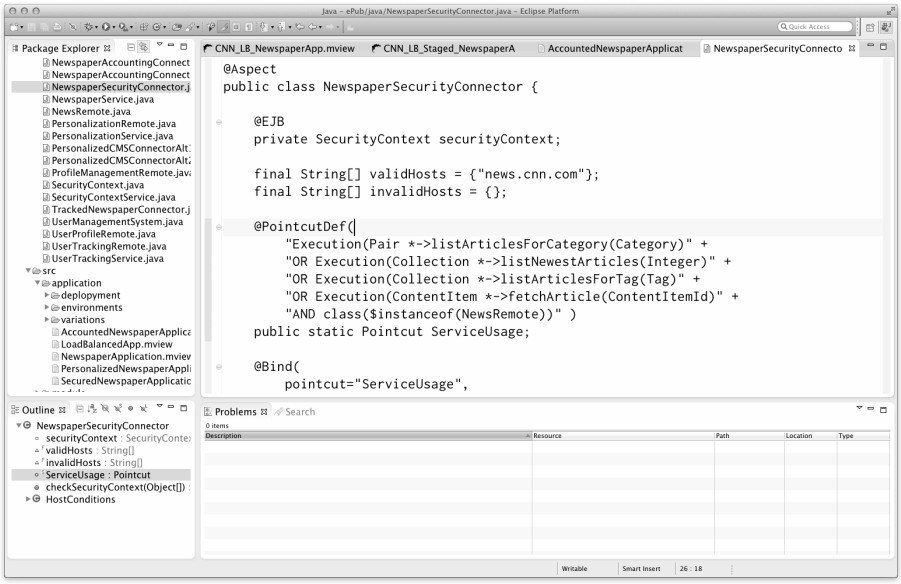


Figure 4.5: Generation of a JBoss Aspect from an MViewADL connector

4.3.4 Middleware Code Generators

In addition to basic tool support for editing MViewADL descriptions that are consistent with the syntax and the semantics of the language, we also provide a prototype code generator for the JBoss middleware. For Spring, we did not repeat the tedious implementation effort of the JBoss prototype, however, in addition to the analysis in Section 4.2, we have constructed a smaller proof-of-concept implementation that confirms the feasibility of a full prototype.

The reason we call the generator a prototype, and not the IDE, is that while the IDE might be rough at the edges, the generator is rough in the middle. Code generation is a complex effort with a lot of edge-cases that needs a robust solution. Our implementation of the generator still lacks that robustness. In addition, there is still information in the MViewADL model that is not considered in the generation. For instance, there is currently no standardized way to include information about allocation in a JBoss application, and our join point model is not completely supported by JBoss AOP —although there are ways around this problem (see Section 4.2.3).

The way in which we handle the transformation of the MViewADL meta-model to that of JBoss, is by building a new AST that is based on the concepts of the target language, from our original AST. This new AST is then passed on as input to a JBossWriter module that walks the AST, producing lines of Java code in a file structure that JBoss understands.

Figure 4.5 shows an example of a JBoss aspect that has been generated from an AO-Connector in MViewADL. We have discussed the specifics of such a transformation in Section 4.2.2, so we will not repeat it here. Our code generation tool supports all the conversion strategies that have been described in Section 4.2. It is able to handle interfaces, methods, components, implicit composition and AO-composition.

4.4 Related Work

In this section we will revisit Table 3.1 of the related work in Chapter 3 and consider the tool and code generation support for each ADLs or approach. In our discussion, we will retain the same categorization into aspect-oriented ADLs, verification and analysis ADLs, and multi-model/view approaches.

Aspect Orientation. DAOP-ADL [PFT05] comes with a set of tools to generate DAOP-ADL descriptions of components and aspects through introspection, and an Eclipse-based IDE to manually produce descriptions. DAOP-ADL was created with the DAOP platform in mind. There is no code-generation support. AO-ADL [PFT11] offers an Eclipse-based specification tool and supports generation of aspects in AspectJ [Asp03] and JBoss AOP. Fractal specifications were created with the FAC platform in mind. It has tools for the specification of descriptions and for querying and reconfiguring the architecture once it has been deployed on the FAC middleware. There is no high-level code generation support. AspectLeda [NPTM09] comes with tools for the specification and verification of descriptions in the language. It supports code generation to Java. AspectualACME [GCB+06] offers no tool support. PRISMA [PACR06] comes with a number of tools for the specification and graphical modelling of PRISMA descriptions. A tool exists that generates C# code.

Verification and Analysis. π-ADL^{ARL} [Oqu04b] comes with tools for graphical and textual modelling. It has tools for verification and architectural refinement.

It supports the generation of Java code. Wright [All97] has a set of preliminary tools for parsing Wright descriptions, and for generating CSP [Plo83] and ACME [GMW97] descriptions. Rapide [Luc96] has a compiler and a tool for graphically browsing the results of an analysis. LEDA [CPT99] does not come with tool support. Darwin [MDK94] has no tool support that we are aware of. SADL [MQR95] supposedly came with a verification tool, but we haven't been able to find or verify this. AADL [FLVC05], as an industry project, comes with extended tool support that includes tools for description, and graphical representation and has support for constrain query languages. In addition AADL supports generation to various forms of ADA.

Multi-model and Multi-view. View Composition [Bou09] defines its ADL as an extension of xADL. It comes with proof-of-concept implementation to automate composition in xADL. Multi Perspective [Gru00] comes with an IDE for description and validation, and it supports code generation to multiple platforms. Stratified Frameworks [AK03] and the Viewpoints Framework [NKF03] do not have tool support.

Summary. Of the aspect-oriented languages, most offer at least some kind of tool support. Some are tied directly into a middleware framework, others offer code generation to a kind of Java (AspectJ is a popular target). Only AO-ADL addresses the goal to generate to multiple design and implementation languages. Many verification and analysis languages do not offer tool support or come with a proof-of-concept analysis or code generation tool. The two more recent ADLs π-ADL^{ARL} and AADL do offer a respectable tool set. The multi-view and multi-model approaches often consider architecture at a highly abstract and conceptual level, making effective tool support harder to realise.

4.5 Validation

The proof of the pudding is in the eating. As tool support was an important goal early on, our tool set was developed and used throughout this work. Our tool evolved continuously in line with the extensions and refinements of the MViewADL model and the ReVew technique. As explained in the Approach section in the introduction of this dissertation, we used our tool support to quickly validate or disprove new ideas in the context of our case study on e-Media.

We have successfully demonstrated our tool at various demo sessions and presentations at internal symposiums. Our tool was also used in the construction of the validations of Chapter 2 and 3. In Chapter 2 we used our tool to create the architecture description of the e-Media case study. From these descriptions, we generated a JBoss implementation that we validated by plugging the source files into the existing e-Media code base and verifying whether the application still worked. In Chapter 3, we used our tool to describe a variability scenario consisting of eight different version of the e-Media architecture. These variations all employed refinement in interesting ways to avoid description duplication, which our tool handled without problems.

4.6 Conclusion

Having tool support is an important characteristic of any architecture description language. With MViewADL, our goal was to provide IDE support that assists architects in the production of description in our language. This chapter presented a high-level overview of the core building blocks of this tool (Section 4.3.4. We discussed the modules that we designed and implemented ourselves, and those modules that are based on reusable frameworks.

Our second goal in this chapter focussed on increasing the value of MViewADL descriptions by supporting code generation to multiple middleware frameworks (i.e. JBoss and Spring). This chapter includes a description of how to realise MViewADL language features and elements in those middleware platforms (Section 4.2). In addition, we briefly discuss how we implemented code generation in our tool support.

The validation of our tool support consists of extended use throughout the refinement of our work, and several live demonstrations. Our discussion of the related work shows that proper tool support is uncommon in an academic context.

Our tool-support has the typical limitation of being specifically tied to the the language in its current form and the set of target languages for code generation that is currently supported. Changes to MViewADL and adding additional target technologies requires changing and redistributing the tool.

The biggest limitation of our tool is that we have been unable to use all the available knowledge in MViewADL models when generating code. The main reason for this limitation is a lack of standardization in middleware frameworks on allocation and aspect orientation.

CHAPTER 5

Conclusion

The development of a distributed system is a complex task that benefits significantly from a properly-designed and well-documented software architecture. During the creation of such an architecture, architects need to address many concerns put forward by a variety of developer stakeholders. To manage the complexity, architects focus their efforts by considering the architecture from the perspective of multiple architectural views [BCK13].

To document the architecture, architects typically make use of one or more architecture description languages (ADL). These ADLs range from simple notations, focussed on communication (e.g. the UML) to formal notations, that target the analysis or expression of a specific system characteristic (e.g. deadlock detection in Wright) [MT00].

However, reflection within the architecture community [MT00, MDT07, MLM+13] has uncovered that, for an ADL to be useful to the architect, it should target a balanced mix of both simple enough to be understood by the stakeholder, yet formal enough to support (automated) analysis and code generation. It should come with features such as dedicated tool support that integrates well in the existing development environment, support for *iterative architecting*, and support for multiple architectural views [MLM+13]. Iterative architecting, as used by Malavolta, et al. [MLM+13], is generally understood as the process where a first

draft is created of the architecture description, that is later on refined during a series of iterative steps.

The goal of this dissertation is to assist developers —architects, coders, and deployers alike— in the creation of complex distributed systems. More specifically, we agree with the above conclusion that the state-of-the-art in architecture description can evolve to better embrace the concept of architectural views; that support for iterative development of architecture description would better match the architectural process; and that tool support is essential for architects to ease the use of an ADL.

5.1 Contributions

The goal that we have put forward, has resulted in the following three contributions:

1. **A multi-view ADL for distributed systems.** We have created MViewADL, an ADL focussing on the description of distributed systems. The ADL is characterized by the following sets of features.

   First, MViewADL is able to capture the contributions of various architectural experts by supporting architecture description in terms of the common architectural views of module, component-and-connector, and allocation. The ADL allows the definition of interfaces, components, and connectors. Connectors compose components on the basis of interface dependency or AO-style interception, and they can adapt between components with mismatching interfaces. It supports the definition of component and connector instances and the assembly of instances into applications. It allows the allocation of instances onto the abstract or concrete hosts of a distributed topology. The related work on ADLs for distributed systems is too often focussed on a single view and particularly allocation is underserved.

   Second, MViewADL supports the concepts and objectives of these architectural views in such a way that it allows the creation of a single model that integrates the concepts of multiple views. For example, the connector supports composition based on module, runtime and allocation context; the application element supports the definition of an abstract topology that can be refined into a physical one. This is in contrast with the related work, where the models are exclusively tied to a single view. The advantage of an

integrated model is that it avoids inconsistencies between complementary models.

Third, the ADL further targets distributed systems by employing loose coupling between heterogeneous components, based on interface dependency. It allows architecture descriptions to scale by means of the implicit connector —only ambiguous connectors, adaptor connectors, and expressive (AO) connectors need to be specified.

We have validated MViewADL in a complex, industry-grade case study that has been developed from requirements to implementation. In addition, code generation support for our language allowed us to replace existing code with generated code without breaking the implementation.

2. **Stepwise refinement in ADLs.**

   Designing the software architecture of a distributed system is a highly iterative process. Requirements engineering results in a prioritized set of requirements that cannot all be handled at one time. Step by step, architects tackle additional requirements and postponed design decisions. This process is known as iterative architecting in software architectures. Currently, this results in monolithic models that have been contributed to by multiple architects with varying expertise (multiple views) at various times.

   Ideally, architecture description acknowledges this iterative process and offers some kind of support in the language to assist architects in the creation of models that can be refined over time. Our study of the related work has shown a lack of support for this process. This results in architecture descriptions that cannot be easily traced to the requirements, that impede on architectural variability, and that are harder to reuse.

   We have proposed the concept of Multi-view Refinement for the modularization of architect contributions to the architecture description of a distributed system. The concept is inspired by the generic idea of stepwise refinement and applies it to architecture description. One model, or architect contribution, is able to refine an existing model by redefining specific parts. We have implemented this concept in a generic technique, called ReVew, that can be applied to various ADLs. The technique applies OO-inheritance to architecture description and supports the redefinition semantics of override and merge.

   One example of a common redefinition is the deployment of the system into another environment. It is also possible to use the technique to define alternative architectures that differ functionally (a newspaper service with

or without user accounting) or qualitatively (a newspaper service with or without load balancing), with optimal reuse of descriptions between alternatives.

We have evaluated ReVew in a modularity and variability validation that compares our technique to two generic refinement techniques (*import* and *flatten*). We show that ReVew bests the other approaches in both validations in terms of size, and the modularity and variability of the architecture description.

3. **ADL tool support and code generation.** In order to make the use of an ADL worthwhile, it needs the proper tool support to assist architects in the production of qualitative descriptions. Furthermore, architecture description is more valuable if it can assist in the production of design or implementation artefacts, that are not tied to a single platform or language.

   In this work, we have provided tool support for both purposes. First, we have created an IDE, on top of Eclipse, that assists in the production of MView-ADL and ReVew architectural descriptions. Second, we have constructed a prototype code generator that produces implementation artefacts tailored to JBoss, and a proof-of-concept that confirms the feasibility of generating artefacts for Spring. In addition, to increase the amount of useful code that can be generated for these platforms, we have proposed a number of patterns to work around missing features in JBoss and Spring.

Together, these contributions realise our goal of assisting developers in the creation of complex distributed systems. We have shown and validated that MView-ADL supports architectural concepts, such as the interface, component, connector, configuration, instance, host, from the various views in the architecture description of a distributed system (Chapter 2). We have shown and validated that code generation is able to turn the architecture description into code artefacts that have been successfully plugged into an existing code base (Chapter 4). We have validated that ReVew reduces architect effort in terms of description size, that it reduces the ripple effects of changes in the architecture by means of better modularity, and that it improves variability through better reuse of description artefacts (Chapter 3).

5.2 Future Work

The future work consists of the short term improvement and long term evolution of MViewADL, as well as the refinement of tool support.

Improving MViewADL

Every language is open to improvement. The improvements we have in mind for MViewADL are short-term: features that we have not been able to fully release within the time-frame of the PhD, and long-term: improvements that would extend or shift the goal of the language.

Features. Some of the short-term features are the hierarchical composite component, immutability and visibility modifiers such as `final` and `private`, and architectural and implementation constraints. However, before implementing any of these features, we need to make sure whether they fit within the language and its goals. For example, composite components are a popular concept with ADLs, but they are not common in the component models of popular Middleware. Can they exist in architecture without having a representation in implementation? Immutability and visibility modifiers are available in most OO programming languages, but it is not yet clear whether they fit within the process of architecture description. On the topic of constraints, MViewADL currently has a number of built-in language constraints that are imposed on architects. However, it could be useful for architects to specify their own constraints, either at the level of architecture or directed at the implementers of the system. For example, mutually exclusive components, a minimal or maximal amount of component instances per back-end instance, or timing constraints for specific method calls.

Behavioural. In its current incarnation, MViewADL focusses on the structural aspects of software architecture —although arguably its AO-composition descriptions are behavioural. There are different ways of adding support for behavioural description to the language. In the connector, behavioural specifications can be used to increase the semantics of composition. This would allow an architect to clearly communicate the meaning of any composition within the model itself, instead of having to revert to less formal means of documentation. Similarly, in the component, behavioural specification allow for a more thourough description of the behaviour of the component.

The related work in architecture description languages contains multiple examples of languages with a behavioural focus, including π-ADL, LEDA, and Wright, Darwin. The former two use π-calculus as the formal modelling language. While a powerful way to express any possible kind of behaviour, using such languages undeniably takes away from the ability to be understood by a wide range of stakeholders. Furthermore, it is far from guaranteed that developers will be able to understand the behavioural specifications without specific training.

Evolving MViewADL

Currently, MViewADL supports the description of a flat topology of hosts. Yet, deployment topologies of distributed applications can be more complex. For instance in cloud environments (IaaS), deployment on virtualized hosts may introduce additional constraints or increase flexibility. PaaS and SaaS infrastructures, on the other hand, completely transform the allocation model. It would be interesting to consider how these new environments impact MViewADL.

Another interesting topic is the goal of retaining more architectural knowledge when going from architecture to design and implementation. In Chapter 4 we have discussed instances where middleware platforms lacked the ability to configure, code, or otherwise describe certain architectural knowledge. We were surprised that some of the lacking abilities had to do with the allocation of components. However, solving these issues will involve improving the state-of-practice in middleware platforms and tool support. This is beyond the responsibility of ADLs.

What we propose on the topic of retaining additional architectural knowledge, however, needs to be addressed in the ADL as well as the subsequent platform and tool chain. Failure and recovery is one of the important concerns in distributed systems. More specifically, what happens to the system when a particular or several components or hosts fail? There are many answers to these questions, and the exact answer depends on the context. One answer is graceful degradation. It is a fault tolerance technique that allows a system to continue proper operation under acceptable quality levels, while working to recover from the failure. However, to recover, the system needs a strategy and the knowledge to determine acceptable operation levels. Architects could use MViewADL to architect several allocation alternatives that can inform the reconfiguration of the system in the event of failure.

Improving Tool Support

As mentioned in the previous point, the state-of-practice middleware platforms lacks support to deal with allocation at the appropriate level of abstraction. Deployers —operators— are left with a set of generic configuration management tools and custom scripting to assist them in managing the infrastructure [BKM+04, DJV10]. Consequently, there is no generic way to generate allocation data from MViewADL descriptions that is immediately useful to operators. Operator intervention or customization of the tool chain is still required. Research into generic configuration management such as that by Rodesek [Rod03], Eilam, et al. [EKK+06], and Van Brabant, et al. [VJ13], are potentially interesting targets for the allocation knowledge in MViewADL descriptions.

APPENDIX A

Syntax of MViewADL

In this appendix chapter we define the abstract syntax of the elements in the MViewADL language. The chapter is structured as follows:

1. Listing A.1 — Miscellaneous types, modifiers, etc.
2. Listing A.2 — Syntax for single and multiple refinement
3. Listing A.3 — Top-level declarations
4. Listing A.4 — Interface declaration
5. Listing A.5 — Method declaration
6. Listing A.6 — Component declaration
7. Listing A.7 — Component and AO-composition declarations
8. Listing A.8 — Pointcut declaration
9. Listing A.9 — Advice declaration
10. Listing A.10 — Application, Host and Instance declarations

Listing A.1: Syntax of various types and modifiers, the identifier, and the qualified reference

```
voidType
   :  'void'
   ;

type
   :  classOrInterfaceType
   |  primitiveType
   ;

classOrInterfaceType
   :  Identifier typeArguments? ('.' Identifier typeArguments? )*
   ;

typeArguments
   :  '<' typeArgument ( ',' typeArgument )* '>'
   ;

typeArgument
   :  type | '?'  ( ('extends' | 'super') type )?
   ;

primitiveType
   :  'boolean'
   |  'char'
   |  'byte'
   |  'short'
   |  'int'
   |  'long'
   |  'float'
   |  'double'
   ;

// Letter and Digit are technology specific and not further detailed here.
Identifier
   :  Letter(Letter|Digit)*
   ;

qualifiedDeclarationReference
   :  Identifier  ('.' Identifier )*
   ;

negationModifier
   :  '!'
   ;

abstractModifier
   :  'abstract'
   ;

overrideModifier
   :  'override'
   |  'merge'
   ;
```

Listing A.2: Single and multiple refinement

```
// single refinement
refinementDeclarationSingle
   :  'refines' refinementDeclarationBody
   ;

// multiple refinement
refinementDeclaration
   :  refinementDeclarationSingle ( ',' refinementDeclarationBody )*
   ;

refinementDeclarationBody
   :  qualifiedDeclarationReference
   ;
```

Listing A.3: The top-level elements are the interface, component, connector, and application

```
compilationUnit
   :  (  interfaceDeclaration
      |  componentDeclaration
      |  connectorDeclaration
      |  applicationDeclaration
      )*
   ;
```

Listing A.4: The interface declaration consists of methods

```
interfaceDeclaration
   :  'interface' Identifier (refinementDeclaration)?
      interfaceBody
   ;

interfaceBody
   :  '{' interfaceBodyDeclaration* '}'
   ;

interfaceBodyDeclaration
   :  (methodDeclaration ';')
   ;
```

Listing A.5: The method declaration

```
methodDeclaration
   :  methodReturnType Identifier formalParameters
   ;

methodReturnType
   :  voidType
   |  type
   ;

formalParameters
   :  '(' (formalParameterDecls)? ')'
   ;

formalParameterDecls
   :  type Identifier (',' formalParameterDecls )?
   ;
```

Listing A.6: The component declaration consists of dependencies

```
componentDeclaration
   :  (abstractModifier)? 'component' Identifier (refinementDeclaration)?
      componentBody
   ;

componentBody
   :  '{' componentBodyDeclaration* '}'
   ;

componentBodyDeclaration
   :  moduleDependencyDeclaration['require']
   |  moduleDependencyDeclaration['provide']
   ;

moduleDependencyDeclaration[String keyword]
   :  ( overrideModifier )? $keyword
      moduleDependencyBody
   ;

moduleDependencyBody
   :  '{' ( commaSeparatedBodyDecls )? '}'
   ;

commaSeparatedBodyDecls
   :  Identifier (',' commaSeparatedBodyDecls )?
   ;
```

Listing A.7: The connector and optional AO-composition declarations

```
connectorDeclaration
   :  (abstractModifier)? 'connector' Identifier (refinementDeclaration)?
      connectorBody
   ;

connectorBody[Connector element]
   :  '{' connectorBodyDeclaration[$element]* '}'
   ;

connectorBodyDeclaration[Connector element]
   :  aoCompositionDeclaration
   |  moduleDependencyDeclaration['require']
   |  moduleDependencyDeclaration['provide']
   ;

// AO-composition
aoCompositionDeclaration
   :  (abstractModifier)? 'ao-composition' Identifier
        (refinementDeclaration)?
      aoCompositionBody
   ;

aoCompositionBody
   :  '{'   pointcutDeclaration? adviceDeclaration? '}'
   ;

adviceType
   :  'before'
   |  'after'
   |  'around'
   ;

joinPointKind
   :  'execution'
   |  'call'
   ;
```

Listing A.8: The pointcut declaration consists of its kind, signature and optional actor (caller/callee) definitions

```
pointcutDeclaration
   :  'pointcut'
      '{' pointcutBodyDeclaration[$pointcut]* '}'
   ;

pointcutBodyDeclaration
   :  pointcutKindDeclaration | pointcutSignatureDeclaration |
       pointcutActorDeclaration
   ;

pointcutKindDeclaration
   :  'kind' ':' joinPointKind ';'
   ;

pointcutSignatureDeclaration
   :  (overrideModifier)? 'signature' ':' pointcutSignatureBodyDecls ';' ;

pointcutSignatureBodyDecls
   :  pointcutMethodSignatureDecl ( ',' pointcutSignatureBodyDecls )?
   ;

pointcutMethodSignatureDecl
   :  (negationModifier)? (Identifier|'*') (Identifier|'*') '('
       (pointcutMethodSignatureParameterDecls )? ')'
   ;

pointcutMethodSignatureParameterDecls
   : (Identifier|'*') (Identifier)? (','
       pointcutMethodSignatureParameterDecls)?
   ;

pointcutActorDeclaration
   :  ( 'caller' | 'callee' ) '{' pointcutActorBodyDecls* '}'
   ;

pointcutActorBodyDecls
   : (overrideModifier)?
   (  'interface' ':'
   |  'component' ':'
   |  'application' ':'
   |  'instance' ':'
   |  'host' ':'
   ) pointcutActorPropDecls ';'
   ;

pointcutActorPropDecls
   :  (negationModifier)? Identifier ( ',' pointcutActorPropDecls )?
   ;
```

Listing A.9: The advice declaration consists of an advice method, the advice type and optional instance

```
adviceDeclaration
   :  'advice' '{' adviceBodyDeclaration* '}'
   ;

adviceBodyDeclaration
   :  adviceMethodDeclaration
   |  adviceTypeDeclaration
   |  adviceInstanceDeclaration
   ;

adviceMethodDeclaration
   :  'method' ':' methodReferenceDeclaration ';'
   ;

methodReferenceDeclaration
   :  Identifier actualParameters
   ;

actualParameters
   :  '(' (actualParameterDecls)? ')'
   ;

actualParameterDecls
   :  Identifier (',' actualParameterDecls )?
   ;

adviceTypeDeclaration
   :  'type' ':' adviceType ';'
   ;

adviceInstanceDeclaration[Advice advice]
   :  'instance' ':' instanceReferenceDeclaration ';'
   ;

instanceReferenceDeclaration
   :  Identifier
   ;
```

Listing A.10: The application declaration consists of host, instance and inline component and connector declarations

```
applicationDeclaration
   :  (abstractModifier)? 'application' Identifier
      (refinementDeclaration)?
      applicationBody
   ;

applicationBody
   : '{' ( applicationBodyDeclaration )* '}'
   ;

applicationBodyDeclaration
   :  moduleContainerDeclarations
   |  hostDeclaration
   |  instanceDeclaration
   ;

// inline components or connectors
moduleContainerDeclarations
   :  componentDeclaration
   |  connectorDeclaration
   ;

//instance
instanceDeclaration
   :  qualifiedDeclarationReference Identifier 'on' Identifier ';'
   ;

// host
hostDeclaration
   :  'host' Identifier ( 'is' StringLiteral )? ';'
   ;
```

APPENDIX B

Variability and Modularity Evaluation Raw Results

In this appendix chapter we present the detailed raw results of the evaluation of ReVew. The chapter is structured as follows:

1. Section B.1 — Evaluation data for the change scenarios
2. Section B.2 — Evaluation data for the ReVew technique
3. Section B.3 — Evaluation data for the Import technique
4. Section B.4 — Evaluation data for the Flatten technique

B.1 Evaluation data for the change scenarios

change	ripple effect in technique					
	MView		*Import*		*Flatten*	
	artifact	loc	#artifacts	loc	#artifacts	loc
NewsRemote	ServiceUsageConnector	15	ServiceUsageConnector	15	ServiceTrackingConnector	35
	ServiceTrackingConnector	24	ServiceTrackingConnector	35	ServiceSecurityConnector	24
			ServiceSecurityConnector	24	ServiceAccountingConnector	24
			ServiceAccountingConnector	24	LB_NewspaperApp	217
			SecuredNewspaperApp	37	NP_NewspaperApp	77
			PersonalizedNewspaperApp	75	CNN_LB_NewspaperApp.mview	237
			AccountedNewspaperApp	33	CNN_LB_StagedNewspaperApp	245
			LB_AccountedNewspaperApp	64	CNN_NP_NewspaperApp	149
			CNN_LBNewspaperApp	44	CNN_NP_StagedNewspaperApp	156
			NYT_LBNewspaperApp	44	NYT_LB_NewspaperApp	237
			CNN_NP_NewspaperApp	36	NYT_LB_StagedNewspaperApp	245
			NYT_NP_NewspaperApp	36	NYT_NP_NewspaperApp	149
			CNN_LB_StagedNewspaperApp	111	NYT_NP_StagedNewspaperApp	156
			NYT_LB_StagedNewspaperApp	111		
			CNN_NP_StagedNewspaperApp	32		
			NYT_NP_StagedNewspaperApp	32		
total	2	39	16	753	13	1951
NewspaperService	AccountedNewspaperApplication	11	AccountedNewspaperApplication	33	LB_NewspaperApp	217
	StagedNewspaperApp	14	StagedNewspaperApp	14	NP_NewspaperApp	77
	LoadBalancedApp	8	LoadBalancedApp	8	CNN_LB_NewspaperApp.mview	237
	NewspaperApplication	14	NewspaperApplication	14	CNN_LB_StagedNewspaperApp	245
	LoadBalancingConnector	18	LoadBalancingConnector	18	CNN_NP_NewspaperApp	149
			LBAccountedNewspaperApp	64	CNN_NP_StagedNewspaperApp	156
			CNN_LB_StagedNewspaperApp	111	NYT_LB_NewspaperApp	237
			NYT_LB_StagedNewspaperApp	111	NYT_LB_StagedNewspaperApp	245
			CNN_NP_StagedNewspaperApp	32	NYT_NP_NewspaperApp	149
			NYT_NP_StagedNewspaperApp	32	NYT_NP_StagedNewspaperApp	156
					LoadBalancingConnector	18
total	5	65	10	437	11	1886
AccountingServer	CNN_LB_NewspaperApp	24	CNN_LB_NewspaperApp	44	CNN_LB_NewspaperApp.mview	237
	CNN_NP_NewspaperApp	16	CNN_NewspaperApp	36	CNN_LB_StagedNewspaperApp	245
	NYT_LB_NewspaperApp	24	NYT_LB_NewspaperApp	44	CNN_NP_NewspaperApp	149
	NYT_NP_NewspaperApp	16	NYT_NewspaperApp	36	CNN_NP_StagedNewspaperApp	156
					NYT_LB_NewspaperApp	237
					NYT_LB_StagedNewspaperApp	245
					NYT_NP_NewspaperApp	149
					NYT_NP_StagedNewspaperApp	156
total	4	80	4	160	8	1574

B.2 Evaluation data for the ReVew technique

stage	variation	artifact	#loc		#conns		#refines	
Module		Interfaces.mview	150		0		0	
		AccountingComponent.mview	10		0		0	
		AuthenticationComponent.mview	10		0		0	
		ClientComponent.mview	10		0		0	
		ContentManagementComponent.mview	6		0		0	
		LoadBalancingComponent.mview	10		0		0	
		NewsDeskComponent.mview	13		0		0	
		NewspaperComponent.mview	12		0		0	
		PersonalizationComponent.mview	11		0		0	
		SecurityContextComponent.mview	10		0		0	
		UserCredentialsComponent.mview	6		0		0	
		UserManagementComponent.mview	6		0		0	
		UserTrackingComponent.mview	10		0		0	
		LoadBalancingConnector.mview	18		1		0	
		PersonalizedContentConnector.mview	24		1		0	
		ServiceAccountingConnector.mview	13		1		2	
		ServiceSecurityConnector.mview	13		1		2	
		ServiceTrackingConnector.mview	24		1		2	
		ServiceUsageConnector.mview	15		1		0	
		total (connector)	**107**	**15.05%**	**6**	**22.22%**	**6**	**9.23%**
		total (module)	**371**	**52.18%**	**6**	**22.22%**	**6**	**9.23%**
Assembly		LoadBalancedNewspaper.mview	8	1.13%	0	0.00%	1	1.54%
		NewspaperApplication.mview	14	1.97%	0	0.00%	0	0.00%
		AccountedNewspaperApplication.mview	17	2.39%	1	3.70%	3	4.62%
		PersonalizedNewspaperApplication.mview	36	5.06%	2	7.41%	6	9.23%
		SecuredNewspaperApplication.mview	21	2.95%	1	3.70%	3	4.62%
	(variation 1a)	LBAccountedNewspaperApp.mview	34	4.78%	2	7.41%	5	7.69%
		LBNewspaperApp.mview	1	0.14%	0	0.00%	1	1.54%
		LBPersonalizedNewspaperApp.mview	34	4.78%	2	7.41%	5	7.69%
	(variation 1b)	NewspaperNPApp.mview	1	0.14%	0	0.00%	1	1.54%
		total (assembly)	**166**	**23.35%**	**8**	**29.63%**	**25**	**38.46%**
Deployment	(variation 2a)	CNN.mview	8	1.13%	0	0.00%	1	1.54%
		NYT.mview	8	1.13%	0	0.00%	1	1.54%
	(variation 2b)	CNN_LB_NewspaperApp.mview	24	3.38%	1	3.70%	3	4.62%
		NYT_LB_NewspaperApp.mview	24	3.38%	1	3.70%	3	4.62%
		CNN_NewspaperApp.mview	16	2.25%	1	3.70%	3	4.62%
		NYT_NewspaperApp.mview	16	2.25%	1	3.70%	3	4.62%
	(variation 2b)	CNN_LB_Staged_NewspaperApp.mview	22	3.09%	3	11.11%	7	10.77%
		NYT_LB_Staged_NewspaperApp.mview	22	3.09%	3	11.11%	7	10.77%
		CNN_NP_StagedNewspaperApp.mview	10	1.41%	1	3.70%	3	4.62%
		NYT_NP_StagedNewspaperApp.mview	10	1.41%	1	3.70%	3	4.62%
		StagedNewspaperApp.mview	14	1.97%	1	3.70%	0	0.00%
		total (deployment)	**174**	**24.47%**	**13**	**48.15%**	**34**	**52.31%**
	TOTAL	[artifact count: 39]	711		27		65	
Deployment		CNN_LB_NewspaperApp	568		15		34	
(each possible variant on its own, as if it was developed individually)		CNN_LB_Staged_NewspaperApp	604		19		41	
		CNN_NP_NewspaperApp	375		7		17	
		CNN_NP_StagedNewspaperApp	338		9		18	
		NYT_LB_NewspaperApp	467		15		34	
		NYT_LB_Staged_NewspaperApp	604		19		41	
		NYT_NP_NewspaperApp	375		7		17	
		NYT_NP_StagedNewspaperApp	399		9		20	
		average	**466.25**		**12.5**		**27.75**	
Deployer Effort		Overall	174	24.47%	13	48.15%	34	52.31%
(the total effort when looking only at the deployer architects)		CNN_LB_NewspaperApp	32	4.50%	1	3.70%	4	6.15%
		CNN_LB_Staged_NewspaperApp	68	9.56%	5	18.52%	11	16.92%
		CNN_NP_NewspaperApp	24	3.38%	1	3.70%	4	6.15%
		CNN_NP_StagedNewspaperApp	48	6.75%	3	11.11%	7	10.77%
		NYT_LB_NewspaperApp	32	4.50%	1	3.70%	4	6.15%
		NYT_LB_Staged_NewspaperApp	68	9.56%	5	18.52%	11	16.92%
		NYT_NP_NewspaperApp	24	3.38%	1	3.70%	4	6.15%
		NYT_NP_StagedNewspaperApp	48	6.75%	3	11.11%	7	10.77%

B.3 Evaluation data for the Import technique

stage	variation	artifact	#loc		#conns		#imports	
Module		Interfaces.mview	150		0		0	
		AccountingComponent.mview	10		0		0	
		AuthenticationComponent.mview	10		0		0	
		ClientComponent.mview	10		0		0	
		ContentManagementComponent.mview	6		0		0	
		LoadBalancingComponent.mview	10		0		0	
		NewsDeskComponent.mview	13		0		0	
		NewspaperComponent.mview	12		0		0	
		PersonalizationComponent.mview	11		0		0	
		SecurityContextComponent.mview	10		0		0	
		UserCredentialsComponent.mview	6		0		0	
		UserManagementComponent.mview	6		0		0	
		UserTrackingComponent.mview	10		0		0	
		LoadBalancingConnector.mview	18		1		0	
		PersonalizedContentConnector.mview	24		1		0	
		ServiceAccountingConnector.mview	24		1		1	
		ServiceSecurityConnector.mview	24		1		1	
		ServiceTrackingConnector.mview	35		1		1	
		ServiceUsageConnector.mview	15		1		0	
		total (connector)	**140**	**11.94%**	**6**	**22.22%**	**3**	**14.29%**
		total (module)	**404**	**34.44%**	**6**	**22.22%**	**3**	**14.29%**
Assembly		LoadBalancedNewspaper.mview	8	0.68%	0	0.00%	1	4.76%
		NewspaperApplication.mview	14	1.19%	0	0.00%	0	0.00%
		AccountedNewspaperApplication.mview	33	2.81%	1	3.70%	1	4.76%
		PersonalizedNewspaperApplication.mview	75	6.39%	2	7.41%	1	4.76%
		SecuredNewspaperApplication.mview	37	3.15%	1	3.70%	1	4.76%
	(variation 1a)	LBAccountedNewspaperApp.mview	64	5.46%	2	7.41%	1	4.76%
		LBNewspaperApp.mview	1	0.09%	0	0.00%	1	4.76%
		LBPersonalizedNewspaperApp.mview	60	5.12%	2	7.41%	1	4.76%
	(variation 1b)	NewspaperNPApp.mview	1	0.09%	0	0.00%	1	4.76%
		total (assembly)	**293**	**24.98%**	**8**	**29.63%**	**8**	**38.10%**
Deployment	(variation 2a)	CNN.mview	8	0.68%	0	0.00%	1	4.76%
		NYT.mview	8	0.68%	0	0.00%	1	4.76%
	(variation 2b)	CNN_LB_NewspaperApp.mview	44	3.75%	1	3.70%	1	4.76%
		NYT_LB_NewspaperApp.mview	44	3.75%	1	3.70%	1	4.76%
		CNN_NewspaperApp.mview	36	3.07%	1	3.70%	1	4.76%
		NYT_NewspaperApp.mview	36	3.07%	1	3.70%	1	4.76%
	(variation 2b)	CNN_LB_Staged_NewspaperApp.mview	111	9.46%	3	11.11%	1	4.76%
		NYT_LB_Staged_NewspaperApp.mview	111	9.46%	3	11.11%	1	4.76%
		CNN_NP_StagedNewspaperApp.mview	32	2.73%	1	3.70%	1	4.76%
		NYT_NP_StagedNewspaperApp.mview	32	2.73%	1	3.70%	1	4.76%
		StagedNewspaperApp.mview	14	1.19%	1	3.70%	0	0.00%
		total (deployment)	**476**	**40.58%**	**13**	**48.15%**	**10**	**47.62%**
	TOTAL	**[artifact count: 39]**	**1173**		**27**		**21**	
Deployment		CNN_LB_NewspaperApp	748		15		12	
(each possible variant on its own, as if it was developed individually)		CNN_LB_Staged_NewspaperApp	873		19		13	
		CNN_NP_NewspaperApp	460		7		8	
		CNN_NP_StagedNewspaperApp	412		9		10	
		NYT_LB_NewspaperApp	614		15		15	
		NYT_LB_Staged_NewspaperApp	873		19		13	
		NYT_NP_NewspaperApp	460		7		8	
		NYT_NP_StagedNewspaperApp	506		9		9	
		average	**618.2**		**12.5**		**11**	
Deployer Effort		Overall	476	40.58%	13	48.15%	10	47.62%
(the total effort when looking only at the deployer architects)		CNN_LB_NewspaperApp	52	4.43%	1	3.70%	2	9.52%
		CNN_LB_Staged_NewspaperApp	177	15.09%	5	18.52%	3	14.29%
		CNN_NP_NewspaperApp	44	3.75%	1	3.70%	2	9.52%
		CNN_NP_StagedNewspaperApp	90	7.67%	3	11.11%	3	14.29%
		NYT_LB_NewspaperApp	52	4.43%	1	3.70%	2	9.52%
		NYT_LB_Staged_NewspaperApp	177	15.09%	5	18.52%	3	14.29%
		NYT_NP_NewspaperApp	44	3.75%	1	3.70%	2	9.52%
		NYT_NP_StagedNewspaperApp	90	7.67%	3	11.11%	3	14.29%

B.4 Evaluation data for the Flatten technique

stage	variation	artifact	#loc		#conns	
Module		Interfaces.mview	150		0	
		AccountingComponent.mview	10		0	
		AuthenticationComponent.mview	10		0	
		ClientComponent.mview	10		0	
		ContentManagementComponent.mview	6		0	
		LoadBalancingComponent.mview	10		0	
		NewsDeskComponent.mview	13		0	
		NewspaperComponent.mview	12		0	
		PersonalizationComponent.mview	11		0	
		SecurityContextComponent.mview	10		0	
		UserCredentialsComponent.mview	6		0	
		UserManagementComponent.mview	6		0	
		UserTrackingComponent.mview	10		0	
		LoadBalancingConnector.mview	18	0.80%	1	1.75%
		PersonalizedContentConnector.mview	24	1.06%	1	1.75%
		ServiceAccountingConnector.mview	24	1.06%	1	1.75%
		ServiceSecurityConnector.mview	24	1.06%	1	1.75%
		ServiceTrackingConnector.mview	35	1.55%	1	1.75%
		total (connector)	**125**	**5.54%**	**5**	**8.77%**
		total (module)	**389**	**17.24%**	**5**	**8.77%**
Assembly		LB_NewspaperApp	217	9.61%	6	10.53%
		NP_NewspaperApp	77	3.41%	2	3.51%
		total (assembly)	**294**	**13.03%**	**8**	**14.04%**
Deployment		CNN_LB_NewspaperApp.mview	237	10.50%	7	12.28%
		CNN_LB_Staged_NewspaperApp.mview	245	10.86%	7	12.28%
		CNN_NP_NewspaperApp.mview	149	6.60%	4	7.02%
		CNN_NP_StagedNewspaperApp.mview	156	6.91%	4	7.02%
		NYT_LB_NewspaperApp.mview	237	10.50%	7	12.28%
		NYT_LB_Staged_NewspaperApp.mview	245	10.86%	7	12.28%
		NYT_NP_NewspaperApp.mview	149	6.60%	4	7.02%
		NYT_NP_StagedNewspaperApp.mview	156	6.91%	4	7.02%
		total (deployment)	**1574**	**69.74%**	**44**	**77.19%**
	TOTAL	**[artifact count: 28]**	**2257**		**57**	
Deployment *(each possible variant on its own, as if it was developed individually)*		CNN_LB_NewspaperApp	626		12	
		CNN_LB_Staged_NewspaperApp	634		12	
		CNN_NP_NewspaperApp	538		9	
		CNN_NP_StagedNewspaperApp	545		9	
		NYT_LB_NewspaperApp	626		12	
		NYT_LB_Staged_NewspaperApp	634		12	
		NYT_NP_NewspaperApp	538		9	
		NYT_NP_StagedNewspaperApp	545		9	
		average	**585.7**		**10.5**	

Bibliography

[AAM99] Marwan Abi-Antoun and Nenad Medvidović, *Enabling the refinement of a software architecture into a design*, UML'99 — The Unified Modeling Language (Robert France and Bernhard Rumpe, eds.), Lecture Notes in Computer Science, vol. 1723, Springer Berlin Heidelberg, 1999, pp. 17–31.

[ADG98] Robert Allen, Remi Douence, and David Garlan, *Specifying and analyzing dynamic software architectures*, Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98) (Lisbon, Portugal), March 1998.

[AF01] Paul Allen and Stuart Frost, *Planning team roles for cbd*, Addison-Wesley Longman Publishing Co., Inc., 2001.

[AK03] Colin Atkinson and Thomas Kühne, *Aspect-oriented development with stratified frameworks*, IEEE Software **20** (2003), no. 1, 81–89.

[AKLS07] Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake, *Aspect refinement-unifying aop and stepwise refinement*, Journal of Object Technology **6** (2007), no. 9, 13–33.

[All97] Robert Allen, *A formal approach to software architecture*, Ph.D. thesis, Carnegie Mellon, School of Computer Science, January 1997.

[Ara94] Guillermo Arango, *A brief introduction to domain analysis*, Proceedings of the 1994 ACM symposium on Applied computing (New York, NY, USA), SAC '94, ACM, 1994, pp. 42–46.

[Asp03] AspectJ project, *AspectJ — crosscutting objects for better modularity*, http://www.eclipse.org/aspectj/, 2003.

[BB01] Felix Bachmann and Len Bass, *Managing variability in software architectures*, Proceedings of the 2001 symposium on Software reusability: putting software reuse in context (New York, NY, USA), SSR '01, ACM, 2001, pp. 126–132.

[BB05] Marco Antonio Barbosa and Luís Soares Barbosa, *Specifying software connectors*, Proceedings of the First international conference on Theoretical Aspects of Computing (Berlin, Heidelberg), ICTAC'04, Springer-Verlag, 2005, pp. 52–67.

[BBB+98] Roland Balter, Luc Bellissard, Fabienne Boyer, Michel Riveill, and J-Y Vion-Dury, *Architecturing and configuring distributed application with olan*, Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (London, UK, UK), Middleware '98, Springer-Verlag, 1998, pp. 241–256.

[BBC+02] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford, *Documenting Software Architecture: Documenting Behavior*, 2002.

[BCK13] Len Bass, Paul Clements, and Rick Kazman, *Software architecture in practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2013.

[BEJV96] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal, *Domain-specific software architectures for guidance, navigation and control*, International Journal of Software Engineering and Knowledge Engineering **6** (1996), no. 2, 201–227.

[BHT+04] Luciano Baresi, Reiko Heckel, Sebastian Thöne, Daniel Varro, Dániel Varró, and Politecnico Di Milano, *Style-based refinement of dynamic software architectures*, In Proc. 4 th Working IEEE/IFIP Conference on Software Architecture, WICSA4, IEEE, 2004, pp. 155–164.

[BKM+04] Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila A. Takayama, and Madhu Prabaker, *Field studies of computer system administrators: analysis of system management tools and practices*, Proceedings of the 2004 ACM conference on Computer supported cooperative work (New York, NY, USA), CSCW '04, ACM, 2004, pp. 388–395.

[BM07] Alan W. Brown and John A. McDermid, *The art and science of software architecture*, Software Architecture (Flavio Oquendo, ed.), Lecture Notes in Computer Science, vol. 4758, Springer Berlin Heidelberg, 2007, pp. 237–256.

[Bou09] Nelis Boucké, *Composition and relations of architectural models supported by an architectural description language*, Ph.D. thesis, KU Leuven, October 2009.

[BR00] Keith H. Bennett and Václav T. Rajlich, *Software maintenance and evolution: a roadmap*, Proceedings of the Conference on The Future of Software Engineering (New York, NY, USA), ICSE '00, ACM, 2000, pp. 73–87.

[BSJ09] Koen Buyens, Riccardo Scandariato, and Wouter Joosen, *Measuring the interplay of security principles in software architectures*, ESEM, 2009, pp. 554–563.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer, *Scaling step-wise refinement*, Proceedings of the 25th International Conference on Software Engineering (Washington, DC, USA), ICSE '03, IEEE Computer Society, 2003, pp. 187–197.

[BWH+08] Nelis Boucké, Danny Weyns, Rich Hilliard, Tom Holvoet, and Alexander Helleboogh, *Characterizing relations between architectural views*, Lecture Notes in Computer Science, vol. 5292, Springer, September 2008, pp. 66–81.

[BWH10] Nelis Boucké, Danny Weyns, and Tom Holvoet, *Composition of architectural models: Empirical analysis and language support*, The Journal of Systems and Software **83** (2010), no. 11, 2108–2127.

[CABA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali, *Variability management in software product lines: a systematic review*, Proceedings of the 13th International Software Product Line Conference (Pittsburgh, PA, USA), SPLC '09, Carnegie Mellon University, 2009, pp. 81–90.

[CPT99] Carlos Canal, Ernesto Pimentel, and José M. Troya, *Specification and refinement of dynamic software architectures*, Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1) (Deventer, The Netherlands, The Netherlands), WICSA1, Kluwer, B.V., 1999, pp. 107–126.

[Cur05] Edward Curry, *Message-oriented middleware*, pp. 1–28, John Wiley & Sons, Ltd, 2005.

[DJV10] Thomas Delaet, Wouter Joosen, and Bart Vanbrabant, *A survey of system configuration tools*, Proceedings of the 23rd Large Installations Systems Administration (LISA) conference, Usenix association, November 2010, pp. 1–14.

[DL03] Mark Denford and John Leaney, *Architecture-based design of computer based systems*, STRAW'03 Second International Software Requirements to Architectures Workshop, 2003, p. 20.

[dSB12] Lakshitha de Silva and Dharini Balasubramaniam, *Controlling software architecture erosion: A survey*, Journal of Systems and Software **85** (2012), no. 1, 132 – 151, Dynamic Analysis and Testing of Embedded Software.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas, *Goal-directed requirements acquisition*, Science of Computer Programming **20** (1993), no. 1-2, 3–50.

[EKK+06] Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, Giovanni Pacifici, John Pershing, and Aditya Agrawal, *Managing the configuration complexity of distributed applications in internet data centers*, Communications Magazine, IEEE **44** (2006), no. 3, 166–177.

[Eva04] Eric Evans, *Domain-driven design: Tackling complexity in the heart of software*, Addison-Wesley, 2004.

[FLVC05] Peter Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert, *An overview of the sae architecture analysis & design language (aadl) standard*, Architecture Description Languages, vol. 176, Springer Boston, 2005.

[Fow02] Martin Fowler, *Patterns of enterprise application architecture*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[GA11] Matthias Galster and Paris Avgeriou, *Handling variability in software architecture: Problems and implications*, 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2011, pp. 171–180.

[Gar98] David Garlan, *Higher-order connectors*, Proceedings of the Workshop on Compositional Software Architectures, 1998.

[GCB+06] Alessandro Garcia, Christina Chavez, Thais Batista, Claudio Sant'anna, Uirá Kulesza, Awais Rashid, and Carlos Lucena, *On the modular representation of architectural aspects*, Software Architecture (Volker Gruhn and Flavio Oquendo, eds.), LNCS, vol. 4344, Springer Berlin / Heidelberg, 2006, pp. 82–97.

[GH09] Vehbi C. Gungor and Gerhard P. Hancke, *Industrial wireless sensor networks: Challenges, design principles, and technical approaches*, Industrial Electronics, IEEE Transactions on **56** (2009), no. 10, 4258–4265.

[GK01] Stephan Gudmundson and Gregor Kiczales, *Addressing practical software development issues in aspectj with a pointcut interface*, Advanced Separation of Concerns, 2001.

[GMW97] David Garlan, Robert Monroe, and David Wile, *Acme: an architecture description interchange language*, Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON '97, IBM Press, 1997, pp. 7–.

[GoP13] GoPivotal, *The spring enterprise platform*, `http://www.springsource.com`, July 2013.

[Gru00] John C. Grundy, *Multi-perspective specification, design and implementation of components using aspects*, International Journal of Software Engineering and Knowledge Engineering **10** (2000), no. 6.

[GSS+06] William G. Griswold, Kevin J. Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan, *Modular software design with crosscutting interfaces*, IEEE Software **23** (2006), no. 1, 51–60.

[Hil10] Rich Hilliard, *On representing variation*, Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (New York, NY, USA), ECSA '10, ACM, 2010, pp. 312–315.

[HMCP04] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo, *Middleware to support sensor network applications*, Network, IEEE **18** (2004), no. 1, 6–14.

[IBM13] IBM, *Ibm websphere*, `http://www-01.ibm.com/software/websphere`, Aug 2013.

[ISO10] ISO/IEC, *Systems and software engineering - architecture description*, ISO/IEC 42010 standard, draft D8 (2010).

[Jac87] Ivar Jacobson, *Object-oriented development in an industrial environment*, Conference proceedings on Object-oriented programming systems, languages and applications (New York, NY, USA), OOPSLA '87, ACM, 1987, pp. 183–191.

[Jac90] Michael Jackson, *Some complexities in computerbased systems and their implications for system development*, CompEuro '90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering, 1990, pp. 344–351.

[Jac01] Michael A. Jackson, *Problem frames: analysing and structuring software development problems*, ACM Press Books, Addison-Wesley, 2001.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh, *The unified software development process*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Kel06] Diane Kelly, *A study of design characteristics in evolving software using stability as a criterion*, IEEE Trans. Software Eng. **32** (2006), no. 5, 315–329.

[Kel07] Stephen Kell, *Rethinking software connectors*, SYANCO '07: International workshop on Synthesis and analysis of component connectors, ACM, 2007, pp. 1–12.

[Kru95] Philippe Kruchten, *The 4+1 view model of architecture*, IEEE Softw. **12** (1995), no. 6, 42–50.

[Kru03] _____, *The rational unified process: An introduction*, 3 ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lag09] Bert Lagaisse, *A comprehensive integration of AOSD and CBSD concepts in Middleware*, Ph.D. thesis, KU Leuven, December 2009.

[Lam07] Butler Lampson, *Principles for computer system design*, ACM Turing award lectures, ACM, 2007, p. 1992.

[LHBL06] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer, *A disciplined approach to aspect composition*, Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (New York, NY, USA), PEPM '06, ACM, 2006, pp. 68–77.

[LJ06] Bert Lagaisse and Wouter Joosen, *True and transparent distributed composition of aspect-components*, Middleware, 2006, pp. 42–61.

[LN04] Phillip A. Laplante and Colin J. Neill, *The demise of the waterfall model is imminent*, Queue **1** (2004), no. 10, 10–15.

[LS80] Bennet P. Lientz and E. Burton Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley Pub (Sd), 1980.

[Luc96] David C. Luckham, *Rapide: A language and toolset for simulation of distributed systems by partial orderings of events.*, Tech. report, Stanford, CA, USA, 1996.

[Mai98] Neil A. M. Maiden, *Crews-savre: Scenarios for acquiring and validating requirements*, Automated Software Engineering **5** (1998), no. 4, 419–446 (English).

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer, *Specifying distributed software architectures*, Proceedings of the 5th European Software Engineering Conference (London, UK, UK), Springer-Verlag, 1995, pp. 137–153.

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer, *A constructive development environment for parallel and distributed programs*, Configurable Distributed Systems, 1994., Proceedings of 2nd International Workshop on, 1994, pp. 4–14.

[MDT07] Nenad Medvidović, Eric M. Dashofy, and Richard N. Taylor, *Moving architectural description from under the technology lamppost*, Inf. Softw. Technol. **49** (2007), no. 1, 12–31.

[MHF+97] Jacques Meekel, Thomas B. Horton, Robert B. France, Charlie Mellone, and Sajid Dalvi, *From domain models to architecture frameworks*, SSR '97: Proceedings of the 1997 symposium on Software reusability (New York, NY, USA), ACM, 1997, pp. 75–80.

[MJVL+07] Tom Mahieu, Wouter Joosen, Dimitri Van Landuyt, Johan Grégoire, Koen Buyens, and Eddy Truyen, *System requirements on digital newspapers*, CW Reports CW484, KU Leuven, Department of Computer Science, March 2007.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang, *What industry needs from architectural languages: A survey*, Software Engineering, IEEE Transactions on **39** (2013), no. 6, 869–891.

[MQR95] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider, *Correct architecture refinement*, IEEE Trans. Softw. Eng. **21** (1995), no. 4, 356–3.

[MR97a] Nenad Medvidović and David S. Rosenblum, *Domains of concern in software architectures and architecture description languages*, Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (Berkeley, CA, USA), DSL'97, USENIX Association, 1997, pp. 1–15.

[MR97b] Mark Moriconi and Robert A. Riemenschneider, *Introduction to sadl 1.0*, Tech. report, Technical Report SRI-CSL-97-01, SRI International, Computer Science Laboratory, 1997.

[MT00] Nenad Medvidović and Richard N. Taylor, *A classification and comparison framework for software architecture description languages*, IEEE Transactions on Software Engineering **26** (2000), no. 1, 70–93.

[NE00] Bashar Nuseibeh and Steve Easterbrook, *Requirements engineering: a roadmap*, Proceedings of the Conference on The Future of Software Engineering (New York, NY, USA), ICSE '00, ACM, 2000, pp. 35–46.

[Nei80] James Milne Neighbors, *Software construction using components*, Ph.D. thesis, 1980.

[NKF03] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein, *Viewpoints: meaningful relationships are difficult!*, International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 676–681.

[NL03] Colin J. Neill and Phillip A. Laplante, *Requirements engineering: the state of the practice*, Software, IEEE **20** (2003), no. 6, 40–45.

[NPTM09] Amparo Navasa, Miguel A. Pérez-Toledano, and Juan M. Murillo, *An adl dealing with aspects at software architecture stage*, Inf. Softw. Technol. **51** (2009), no. 2, 306–324.

[NSV+06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée, *Explicitly distributed aop using awed*, AOSD'06, ACM, 2006.

[Nus01] Bashar Nuseibeh, *Weaving the software development process between requirements and architecture*, ICSE-2001 International Workshop: From Software Requirements to Architectures (STRAW-01) (Toronto, Canada), September 2001.

[OBS03] Ossama Othman, Jaiganesh Balasubramanian, and Douglas C. Schmidt, *The design of an adaptive middleware load balancing and monitoring service*, Proceedings of the 3rd International Workshop on Self-Adaptive Software, ACM, 2003, pp. 205–213.

[OdbvDLJ12] Steven Op de beeck, Marko van Dooren, Bert Lagaisse, and Wouter Joosen, *Multi-view refinement of ao-connectors in distributed software systems*, Proceedings of the 11th annual international conference on Aspect-oriented Software Development, ACM, March 2012, pp. 251–262.

[OdbVLTJ08] Steven Op de beeck, Dimitri Van Landuyt, Eddy Truyen, and Wouter Joosen, *A domain-specific middleware layer using aosd: next-generation digital news publishing*, Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion, ACM, December 2008, pp. 78–81.

[OG05] Steven Op de beeck and Johan Grégroire, Master's thesis, KU Leuven, 6 2005, In dutch.

[OGTJ06] Steven Op de beeck, Johan Grégoire, Eddy Truyen, and Wouter Joosen, *On the criteria of aspectual component models*, Proceedings of the International Conference on Aspect-Oriented Software Development, 3 2006.

[Oqu04a] Flávio Oquendo, *π-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures*, ACM SIGSOFT Software Engineering Notes **29** (2004), no. 3.

[Oqu04b] Flavio Oquendo, *π-arl: an architecture refinement language for formally modelling the stepwise refinement of software architectures*, SIGSOFT Softw. Eng. Notes **29** (2004).

[Ora13a] Oracle, *Oracle internet application server*, `http://www.oracle.com/technetwork/middleware/ias/overview`, Aug 2013.

[Ora13b] _____, *Oracle soa*, `http://www.oracle.com/us/products/middleware/soa/overview`, Aug 2013.

[OTB+06] Steven Op de beeck, Eddy Truyen, Nelis Boucké, Franciscus Sanen, Maarten Bynens, and Wouter Joosen, Tech. report, 2 2006.

[OvDLJ13] Steven Op de beeck, Marko van Dooren, Bert Lagaisse, and Wouter Joosen, *Modularity and variability of distributed software architectures through multi-view refinement of ao-connectors*, T. Aspect-Oriented Software Development **10** (2013), 109–147.

[OvDLJ14] _____, *e-media case study evaluation*, `http://opdebeeck.org/phd/eval`, Jan 2014.

[PACR06] Jennifer Pérez, Nour Ali, Jose A. Carsí, and Isidro Ramos, *Designing software architectures with an aspect-oriented architecture description language*, Proceedings of the 9th international conference on Component-Based Software Engineering (Berlin, Heidelberg), CBSE'06, Springer-Verlag, 2006, pp. 123–138.

[Par72] David L. Parnas, *On the criteria to be used in decomposing systems into modules*, Commun. ACM **15** (1972), no. 12, 1053–1058.

[PD90] Rubén Prieto-Díaz, *Domain analysis: an introduction*, SIGSOFT Softw. Eng. Notes **15** (1990), no. 2, 47–54.

[PFT05] Mónica Pinto, Lidia Fuentes, and José M. Troya, *A dynamic component and aspect-oriented platform.*, Computer Journal **48** (2005), no. 4.

[PFT11] Mónica Pinto, Lidia Fuentes, and José María Troya, *Specifying aspect-oriented architectures in ao-adl*, Information and Software Technology, Elsevier, 2011.

[Plo83] Gordon Plotkin, *An operational semantics for csp*, Logics of Programs and Their Applications (A. Salwicki, ed.), Lecture Notes in Computer Science, vol. 148, Springer Berlin Heidelberg, 1983, pp. 250–252.

[PQ95] Terence J. Parr and Russell W. Quong, *Antlr: A predicated (k) parser generator*, 1995.

[PRM+03] Jennifer Pérez, Isidro Ramos, Javier Jaén Martínez, Patricio Letelier, and Elena Navarro, *Prisma: Towards quality, aspect oriented and dynamic software architectures*, International Conference on Quality Software, 2003, pp. 59–66.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli, *Jac: an aspect-based distributed dynamic framework*, Software Practicate and Experience **34** (2004), no. 12, 1119–1148.

[PSDC08] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Thierry Coupaye, *A component-based and aspect-oriented model for software evolution*, Int. J. Comput. Appl. Technol. **31** (2008), no. 1/2, 94–105.

[PW92] Dewayne E Perry and Alexander L Wolf, *Foundations for the study of software architecture*, ACM SIGSOFT Software Engineering Notes **17** (1992), no. 4, 40–52.

[Red13a] RedHat, *Jboss appliction server*, `http://www.jboss.org/jbossas`, July 2013.

[Red13b] _____, *Redhat jboss soa platform*, `http://www.redhat.com/products/jbossenterprisemiddleware/soa`, Aug 2013.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The unified modeling language reference manual, second edition*, Addison-Wesley, 2005.

[Rod03] Gabi D. Rodosek, *A generic model for it services and service management*, Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on, 2003, pp. 171–184.

[Roy87] Winston W. Royce, *Managing the development of large software systems: concepts and techniques*, Proceedings of the 9th international conference on Software Engineering (Los Alamitos, CA, USA), ICSE '87, IEEE Computer Society Press, 1987, pp. 328–338.

[RW11] Nick Rozanski and Eóin Woods, *Software systems architecture: Working with stakeholders using viewpoints and perspectives*, Addison-Wesley, 2011.

[SBJ10] Riccardo Scandariato, Koen Buyens, and Wouter Joosen, *Automated detection of least privilege violations in software architectures*, ECSA, 2010, pp. 150–165.

[SFG99] Helen Sharp, Anthony Finkelsteiin, and Galal Galal, *Stakeholder identification in the requirements engineering process*, Proceedings of the 10th International Workshop on Database & Expert Systems Applications (Washington, DC, USA), DEXA '99, IEEE Computer Society, 1999, pp. 387–.

[Sha89] Mary Shaw, *Larger scale systems require higher-level abstractions*, Proceedings of the 5th international workshop on Software specification and design (New York, NY, USA), IWSSD '89, ACM, 1989, pp. 143–146.

[Sha94] _____, *Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status*, Tech. report, Pittsburgh, PA, USA, 1994.

[SMC74] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine, *Structured design*, IBM Systems Journal **13** (1974), no. 2, 115–139.

[Som10] Ian Sommerville, *Software engineering*, 9. ed., Addison-Wesley, Harlow, England, 2010.

[SSA02] Richard E. Schantz, Douglas C. Schmidt, and The Pennsylvania State University CiteSeer Archives, *Middleware for distributed systems - evolving the common structure for network-centric applications*, Encyclopedia of Software Engineering **1** (2002).

[Str00] Bjarne Stroustrup, *The c++ programming language*, 3rd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Szy02] Clemens Szyperski, *Component software: beyond object-oriented programming*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TMD09] Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy, *Software architecture: Foundations, theory, and practice*, Wiley Publishing, 2009.

[TN05] Éric Tanter and Jacques Noyé, *A versatile kernel for multi-language aop*, Generative Programming and Component Engineering, LNCS, Springer Berlin / Heidelberg, 2005.

[vDSJ12] Marko van Dooren, Eric Steegmans, and Wouter Joosen, *An object-oriented framework for aspect-oriented languages*, AOSD, 2012, pp. 215–226.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg, *On the notion of variability in software product lines*, Working IEEE/IFIP Conference on Software Architecture Proceedings., 2001, pp. 45–54.

[VJ13] Bart Vanbrabant and Wouter Joosen, *A framework for integrated configuration management tools*, IFIP/IEEE International Symposium on Integrated Network Management, May 2013.

[VLGM+06] Dimitri Van Landuyt, Johan Grégoire, Sam Michiels, Eddy Truyen, and Wouter Joosen, *Architectural design of a digital publishing system*, CW Reports CW465, Department of Computer Science, KU Leuven, Leuven, Belgium, October 2006.

[VOK+] Dimitri Van Landuyt, Steven Op de beeck, Bas Kemper, Eddy Truyen, and Wouter Joosen, *Building a next-generation digital publishing platform using aosd*, `http://distrinet.cs.kuleuven.be/projects/digitalpublishing`.

[Völ07] Markus Völter, *Software architecture documentation in the real world*, Tutorial. In 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2007.

[VOTJ09] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Wouter Joosen, *Domain-driven discovery of stable abstractions for pointcut interfaces*, Proceedings of the 8th ACM international conference on Aspect-oriented software development (New York, NY, USA), AOSD '09, ACM, 2009, pp. 75–86.

[VOTJ12] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Wouter Joosen, *Domain-driven discovery of stable abstractions for pointcut interfaces*, T. Aspect-Oriented Software Development **9** (2012), 1–52.

[VOTV10] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Petrus Verbaeten, *Building a digital publishing platform using aosd*, LNCS Transactions on Aspect-Oriented Software Development, vol. 8, December 2010.

[WC07] Byron J. Williams and Jeffrey C. Carver, *Characterizing software architecture changes: An initial study*, Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (Washington, DC, USA), ESEM '07, IEEE Computer Society, 2007, pp. 410–419.

[WH05] Eoin Woods and Rich Hilliard, *Architecture description languages in practice session report*, Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on, 2005, pp. 243–246.

[Wir71] Niklaus Wirth, *Program development by stepwise refinement*, Commun. ACM **14** (1971), no. 4, 221–227.

[Wol97] Alexander L. Wolf, *Succeedings of the second international software architecture workshop (isaw-2)*, SIGSOFT Softw. Eng. Notes **22** (1997), no. 1, 42–56.

[Woo05] Eoin Woods, *Architecture description languages and information systems architects: Never the twain shall meet?*, Artechra White paper (2005).

List of Publications

Journal papers

2013 Steven Op de beeck, Marko van Dooren, Bert Lagaisse, Wouter Joosen, *Modularity and variability of distributed software architectures through multi-view refinement of AO-connectors*, LNCS Transactions on Aspect-Oriented Software Development, volume 7800.

2010 Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, Pierre Verbaeten, *Building a digital publishing platform using AOSD*, LNCS Transactions on Aspect-Oriented Software Development, volume 9.

2010 Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, Wouter Joosen, *Domain-driven discovery of stable abstractions for pointcut interfaces*, LNCS Transactions on Aspect-Oriented Software Development, volume 9.

International conference, and workshop papers

2012 Steven Op de beeck, Marko van Dooren, Bert Lagaisse, and Wouter Joosen, *Multi-view refinement of ao-connectors in distributed software systems*, Proceedings of the 11th annual international conference on Aspect-oriented Software Development, ACM, Germany.

2009 Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, Wouter Joosen, *Domain-driven discovery of stable abstractions for reusable pointcut interfaces*, Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2009), ACM, Charlottesville, Virginia, USA.

2008 Steven Op de beeck, Dimitri Van Landuyt, Eddy Truyen, and Wouter Joosen, *A domain-specific middleware layer using aosd: next-generation digital news publishing*, Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion, ACM, Belgium.

2007 Steven Op de beeck, Dimitri Van Landuyt, Johan Grégoire, Riccardo Scandariato, Wouter Joosen, A Jackson, Siobhan Clarke, *Aspectual vs. component-based decomposition: a quantitative study*, Proceedings of the International Conference on Aspect Oriented Software Development, Vancouver, BC.

2006 Steven Op de beeck, Johan Grégoire, Eddy Truyen, Wouter Joosen, *On the criteria of aspectual component models*, Proceedings of the International Conference on Aspect-Oriented Software Development, Germany.

Demonstrations

2008 Middleware, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1.

2009 AOSD, 8th International Conference on Aspect-Oriented Software Development, Charlottesville, Virginia, USA, March 2–6.

2009 Course on Design of Distributed Applications (Ontwerp van Gedistribueerde Applicaties), KU Leuven, Leuven, May.

2009 AOSD Summer School, 4th European Summer School on Aspect-oriented Software Development, Ecole des Mines de Nantes, Nantes, France, August 24–28.

2010 Course on Design of Distributed Applications (Ontwerp van Gedistribueerde Applicaties), KU Leuven, Leuven, May.

2012 DistriNet Research and Development Symposium, KU Leuven, Leuven, April.

Technical reports

2006 Steven Op de beeck, Eddy Truyen, Nelis Boucké, Franciscus Sanen, Maarten Bynens, Wouter Joosen, *A study of aspect-oriented design approaches*, volume CW435, Department of Computer Science, KU Leuven, Leuven, Belgium.

In most people's vocabularies, design means veneer. It's interior decorating. It's the fabric of the curtains and the sofa. But to me, nothing could be further from the meaning of design. Design is the fundamental soul of a human-made creation that ends up expressing itself in successive outer layers of the product or service.

—Steven Paul Jobs

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
SCIENTIFIC COMPUTING GROUP
Celestijnenlaan 200A box 2402
B-3001 Heverlee
steven@opdebeeck.org
http://distrinet.cs.kuleuven.be